Principles and solutions for improved availability and code vulnerability detection

Atiiq, Syafiq Al

2025

Document Version:
Peer reviewed version (aka post-print)

Principles and Solutions for Improved Availability and Code Vulnerability Detection

Syafiq Al Atiiq

LUND
UNIVERSITY

ISBN 978-91-8104-521-5 (printed)
ISBN 978-91-8104-522-2 (electronic)
Series of licentiate and doctoral theses
No. 185
ISSN 1654-790X-185

Syafiq Al Atiiq
Department of Electrical and Information Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

Typeset using LaTeX.
Printed in Sweden by Tryckeriet i E-huset, Lund, 2025.

Abstract

In a time when digital services are fundamental to everyday living, ensuring the reliability of these systems is crucial. Availability, the property of a system being accessible and usable upon demand, is a key aspect of reliability. This dissertation addresses three domains where availability is critical: (i) Internet of Things (IoT), (ii) 5G networks (with particular focus on the data analytics component), and (iii) software vulnerability detection. Although each domain poses unique challenges, ranging from resource-constrained IoT devices to large-scale 5G networks and massive software codebases, they share a common requirement: maintaining dependable, uninterrupted operations despite failures and attacks.

First, this work proposes new techniques to counteract Denial of Service (DoS) attacks targeting IoT systems, where IoT devices can be either victims or potential attack sources orchestrated by a centralized adversary. Efficient, collaborative approaches and lightweight detection mechanisms are shown to reduce computational burden and energy consumption on resource-constrained IoT devices while preserving their service. Additionally, a collaborative approach is demonstrated to mitigate the detrimental effects of DoS attacks on victims, such as service disruption or resource exhaustion, irrespective of the victim's computing capabilities (i.e., a high-performance server vs. a small embedded device). Furthermore, we extend existing IoT device recovery schemes claimed to provide guaranteed recovery from a compromised device. We identify limitations in these schemes and propose enhancements to better handle runtime software attacks and network-level disruptions, thereby strengthening the IoT devices' ability to recover from a compromised state.

Second, the research explores the availability challenges in 5G by focusing on the Network Data Analytics Function (NWDAF), which uses machine learning for tasks such as mobility prediction. Mobility prediction is crucial in 5G as it enables the network to anticipate the movement of devices and proactively allocate resources at the predicted location. This helps ensure a seamless handover process and maintains uninterrupted service for users. The impact of attacks on the NWDAF's mobility prediction is investigated, considering two types of attacks: inference-time attacks, where malicious devices provide false location data to mislead the trained model, and training-time attacks, where attackers manipulate the training data to degrade the model's performance. For the former, the results demonstrate that even a small number of compromised devices participating in these attacks can substantially reduce the accuracy of the NWDAF's mobility predictions. For the latter, we investigate strategies for robust retraining and model selection, particularly through Automatic Machine Learning (AutoML), showing that initial selection with a long time-budget is more robust than reselecting the model for each retraining.

Finally, the dissertation investigates the use of Code Language Models (Code-LMs) for software vulnerability detection across multiple programming languages,

including C/C++, JavaScript, PHP, Java, Python, and Go. Results show that for C/C++ code, models trained and tested on a single vulnerability type achieve higher recall and F1 scores for that specific type. Also, certain vulnerability types are found to share detectable patterns that enable some cross-type generalization, while others have more unique representations that do not transfer well. As a continuation of code-LM work on C/C++, comparing vulnerability detection performance across languages reveals that models generally perform better on non-C/C++ languages, with some achieving meaningfully higher F1 scores compared to prior work on C/C++. However, performance still varies substantially between languages. To understand the variations further, we analyze the relationship between code complexity metrics and detection performance (F1) and find only weak and inconsistent correlations.

Acknowledgements

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

I would like to express my sincere gratitude to my supervisor, Christian Gehrmann, for his support, guidance, and mentorship. Your feedback and thought-provoking discussions have been instrumental in shaping my research and helping me grow as a scholar. Thank you for always making time for me despite your busy schedule. Your impact on my academic development will undoubtedly extend far beyond the pages of this dissertation.

Collaborating on research and papers with my brilliant co-authors, Jakob, Karim, Kevin, Luis, and Yachao, has been truly enjoyable. Each of you has contributed uniquely to this journey, and I look forward to future collaborations.

To my office mate Vu, a true comrade-in-arms, thank you for your brutal honesty and the countless moments of laughter (and stress) we've shared. I also want to give a shout-out to the rest of the security group: Alex, Arthur, Dachao, Denis, Elena, Erik, Hui, Jing, Joakim, Jonathan, Linus, Maggie, Markus, Martin G, Martin H, Mohsen, Mustafa, Paul, Pegah, Qian, Rohon, Shouran, and Thomas. Thank you for being awesome colleagues and for all the fun times we've shared during lunches and fika breaks.

To the Indonesian community in Skåne, thank you for keeping me sane during the dark winters. Special thanks to Andri, Desna, Lintang, and Ayu, who have become an extended family even though we're miles away from our homeland.

To my loving mother, Ibuk Mukarromah, your endless love and prayers have been a constant source of strength. To my three brothers, Mahuddin, Alan, and Yusron, thank you for introducing me to the world of engineering and for being my role models. To my late father, Bapak Muhadjir, a man with a heart of gold, this dissertation is as much yours as it is mine; I deeply missed all the silly conversations and the random "let's-take-the-motorcycle-and-go-somewhere" moments we had.

To my kids, Khadija Sabrina and Ahmad Fahruddin, your smiles, hugs, and innocent curiosity have been a constant reminder of what truly matters in life. Thank you for understanding when Daddy had to work long hours and for always welcoming me home with open arms.

Saving the best for last, my deepest gratitude to my better half, Amirinnisa, for her unwavering love and patience throughout this journey. Your belief in me has been my anchor, keeping me grounded even when I felt like I was floating adrift. Thank you for being my personal superhero, always ready to swoop in, and for the occasional reminder that sleep is not optional. I couldn't have done this without you. ILYTTMAB!

Syafiq
Lund, March 2025

Contribution Statement

This dissertation includes the following papers:

Paper I Syafiq Al Atiiq, Christian Gehrmann. "CLI-DOS: Collaborative Counteraction against Denial of Service in the Internet of Things". In 18^{th} *International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2020, Austin, TX, USA*. pp. 1-6, IEEE.

Paper II Syafiq Al Atiiq, Christian Gehrmann. "X-Pro: Distributed XDP Proxies Against Botnets of Things". In 26^{th} *Nordic Conference on Secure IT Systems, Nordsec 2021, Tampere, Finland*. LNCS Vol. 13115, pp. 51–71, Springer.

Paper III Syafiq Al Atiiq, Christian Gehrmann. "Regaining Dominance in CIDER and Lazarus". In *IEEE Access, 2024*. vol. 12, pp. 124589-124603, IEEE.

Paper IV Syafiq Al Atiiq, Yachao Yuan, Christian Gehrmann, Jakob Sternby, Luis Barriga. "Attacks Against Mobility Prediction in 5G Networks". In 22^{nd} *International Conference on Trust, Security, and Privacy in Computing and Communications, TrustCom 2023, Exeter, United Kingdom*. pp. 1502-1511, IEEE.

Paper V Syafiq Al Atiiq, Christian Gehrmann, Yachao Yuan, Jakob Sternby. "AutoML in the Face of Adversity: Securing Mobility Predictions in NWDAF". In 9^{th} *International Conference on Fog and Mobile Edge Computing, FMEC 2024, Malmö, Sweden*. pp. 90-98, IEEE.

Paper VI Syafiq Al Atiiq, Christian Gehrmann, Karim Khalil, Kevin Dahlén. "Catching Common Vulnerabilities with Code Language Models". Submitted to 20^{th} *International Conference on Availability, Reliability and Security, ARES 2025, Ghent, Belgium*.

Paper VII Syafiq Al Atiiq, Christian Gehrmann, Kevin Dahlén. "Vulnerability Detection in Popular Programming Languages with Language Models". Submitted to 20^{th} *International Conference on Availability, Reliability and Security, ARES 2025, Ghent, Belgium*.

Below is a table that outlines Syafiq Al Atiiq's responsibilities in each paper.

Paper	Writing	Concepts	Implementation	Evaluation
I	★★★★	–	★★★★★	★★★★★
II	★★★★★	★★★★	★★★★★	★★★★★
III	★★★★	★★	★★★★★	★★★★★
IV	★★★★	★★★	★★★★★	★★★★★
V	★★★★★	★★★★★	★★★★★	★★★★★
VI	★★★★	★★★★	★★★★★	★★★★★
VII	★★★★★	★★★★★	★★★★★	★★★★★

Legend:
★★★★★ = Lead (primary responsibility)
★★★★ = Major
★★★ = Significant
★★ = Moderate
★ = Minor
– = None

Table 1: Contributions per Paper.

Syafiq's contributions are explained in more detail in the following paragraphs.

In Paper I, Syafiq was the developer responsible for building the proof-of-concept (PoC) implementation and performing the evaluation. Syafiq also made major contributions to the writing of the paper.

In Paper II, Syafiq developed the idea of using kernel networking for the proxy model and was also the main developer responsible for building the PoC implementation and conducting the evaluation. Syafiq took the lead and was the primary person responsible for the writing.

In Paper III, Syafiq developed the attack (both the concept and implementation) and contributed equally to the concept of the proposed solution. Syafiq also developed the PoC implementation of the proposed solution. And last, he contributed equally to the writing of the paper.

In Paper IV, Syafiq wrote an extension for the simulator to enable it to produce adversarial mobility patterns for attacking the existing legitimate mobility. Syafiq performed the attack, evaluation, and analysis using this extended simulator. Syafiq was the main contributor to the paper's writing.

In Paper V, Syafiq developed the initial idea and concept for the paper. Syafiq was responsible for both the implementation and evaluation aspects of the work.

In Paper VI, Syafiq was involved in the ideation process to develop the idea and concept for the paper. Syafiq was responsible for a major portion of the writing

and the lead responsible for the implementation.

In Paper VII, Syafiq acted as the lead writer for the paper, playing a key role in the ideation process to shape the idea and concept. Syafiq also took charge of its implementation.

Section 3.1 further elaborates on Syafiq's specific contributions *to the research field* in each of the included papers.

Other Contributions

The following peer-reviewed publications are not incorporated into the main body of this dissertation:

- Marco Tiloca, Rikard Höglund, Syafiq Al Atiiq. "SARDOS: Self-Adaptive Reaction Against Denial of Service in the Internet of Things," Peer-reviewed in 5^{th} *International Conference on Internet of Things: Systems, Management, and Security, IoTSMS 2018, Valencia, Spain*, pp. 54-61, IEEE.

- Syafiq Al Atiiq, Aris Cahyadi Risdianto. "Demystifying AMD SEV Performance Penalty for NFV Deployment," Peer-reviewed in 13^{th} *International Conference on Networks, Communication, and Computing, ICNCC 2024, Bangkok, Thailand*, ACM.

- Syafiq Al Atiiq, Christian Gehrmann, Karim Khalil, Jakob Sternby, Yachao Yuan. "Resilient Automatic Model Selection for Mobility Prediction," Submitted to the Journal of Cluster Computing.

Contents

Introduction

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> — Leslie Lamport

Imagine a day when your mobile phone suddenly decides it needs a break, precisely the moment you most need it. To compound the frustration, your internet connection also goes offline, so you cannot even run a quick search on "*how to fix a phone that won't turn on*" from another device. It feels like trying to find a light switch in a pitch-black room: you know the solution is out there, but you have no way of reaching it. Even though the two breakdowns may be unrelated, their simultaneous failure shows how dependent we have become on these systems to manage our everyday lives.

This reliance on connectivity was also shown during the COVID-19 pandemic [Fel+20; OZ23]. As lockdowns and social distancing measures took effect worldwide, the internet evolved into an indispensable gateway for work, education, and social interaction. These online networks effectively became our collective safety net, enabling vital research to continue (research that includes some of the work presented in this dissertation) and showing just how dependent we have become on the seamless operation of these modern systems.

But what exactly is a **system**? A system, as defined by [Mea08], is "*a set of interconnected components designed to work collectively toward achieving a specific goal.*" While this concept can be interpreted broadly across various contexts, this dissertation focuses specifically on systems within three domains: the Internet of Things (IoT), the 5^{th} generation of the cellular network (5G), and software verification processes. In each of these domains, ensuring that the interconnected components continuously function as intended is crucial for both everyday operations and broader societal needs.

Building on the previous understanding, this dissertation explores how **avail-**

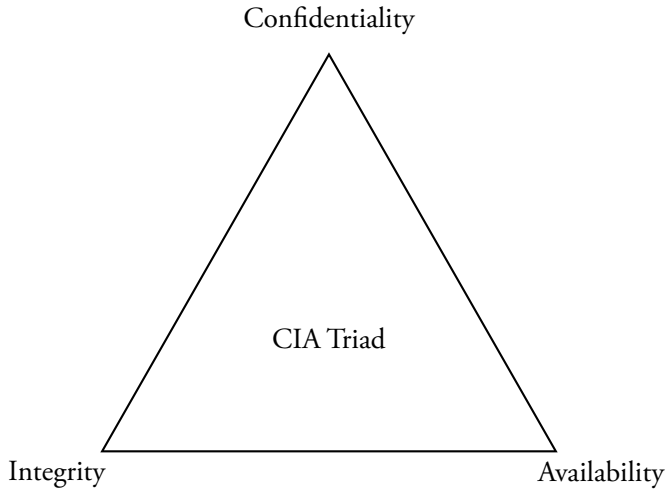

Figure 1.1: The CIA Triad: Confidentiality, Integrity, and Availability

ability issues affect IoT and 5G systems while also exploring how vulnerabilities in software can be caught early. Availability refers to the property of a system being accessible and usable upon demand by an authorized entity [Avi+04]. In other words, it is the assurance that a system will work as intended and when needed.

Availability is one of the three core principles of information security, alongside confidentiality and integrity, collectively forming the Confidentiality, Integrity, and Availability (CIA) triad[1] as shown in figure 1.1:

- **Confidentiality**: Ensuring that sensitive data remains private and accessible only to authorized individuals.
- **Integrity**: Preventing unauthorized alteration or tampering of data or systems to maintain their accuracy and trustworthiness.
- **Availability**: Ensuring that systems, applications, and data are continuously accessible to legitimate users when needed.

Consider a hospital's IoT-enabled patient monitoring system[2] [CS24]: Confidentiality protects sensitive health data, Integrity ensures medication dosages aren't tampered with, and Availability guarantees life-critical alerts always reach nurses. This real-world dependency on the CIA triad drives its adoption across industries where system failures have tangible consequences.

In the context of IoT and 5G, availability issues can arise from various factors such as device failures [KRS09], network congestion [Wel05], or malicious

[1] https://www.nccoe.nist.gov/publication/1800-26/VolA/index.html
[2] https://tele2iot.com/article/remote-patient-monitoring/

attacks like Denial of Service (DoS) [Nee93]. These disruptions can have severe consequences, especially as these technologies become more integrated into critical infrastructure and everyday life. For example, vulnerabilities in medical IoT devices, such as wireless pacemakers, could expose patients to life-threatening cyberattacks[3], while 5G network outages could disrupt emergency services[4].

Starting with IoT, this dissertation explores the challenges posed by DoS attacks. When multiple IoT devices are compromised and become attackers, the attack's magnitude can be amplified enormously, as shown by the Mirai botnet attack [Ant+17a]. Conversely, when an IoT device becomes a victim, the consequences can be severe due to its resource-constrained nature [GTH15]. Also, this dissertation proposes an efficient collaborative protocol to detect and mitigate DoS attacks on IoT devices and identifies potential availability issues in existing recovery architectures, CIDER [Xu+19] and Lazarus [Hub+20], offering an updated design.

Moving to 5G, the research focuses on the Network Data Analytics Function (NWDAF) [3GP22c], a core network node designed to optimize network operations through data analysis. We examine the impact of adversarial data contamination on the machine learning (ML) models NWDAF uses, particularly for the use case of mobility prediction. Using data from Ericsson's Global Artificial Intelligence Accelerator (GAIA) platform[5], we extend the tool to generate adversarial patterns. We investigate the model's reaction to contaminated data, also considering the use of Automated Machine Learning (AutoML) [HKV19], an automatic mechanism of selecting, optimizing, and evaluating ML models for model discovery.

While availability is central to the IoT and 5G, this dissertation also explores vulnerability detection in software, which is related but distinct. Software vulnerabilities pose a significant threat to system security. Attackers can exploit flaws in the design or implementation of software to crash systems, leak sensitive data, or gain unauthorized control. Therefore, effective vulnerability detection is crucial for maintaining the security and reliability of systems.

In the software verification domain, we investigate Code Language Model (code-LM)-based vulnerability detection systems [Zha+24]. Despite outperforming previous approaches, these systems have not achieved satisfactory performance for the C/C++ programming language. We analyze the most common vulnerability representations to identify patterns and potential improvements. Additionally, recognizing the focus on C/C++, the dissertation evaluates the performance of

[3] https://www.science.org/content/article/could-wireless-pacemaker-let-hackers-take-control-your-heart

[4] https://news.sky.com/story/widespread-disruption-in-denmark-after-mobile-network-outage-13262696

[5] https://www.ericsson.com/en/about-us/company-facts/ericsson-worldwide/india/ericsson-in-india/global-artificial-intelligence-accelerator-india

Code-LMs on other popular programming languages to address this gap in the current literature.

Our modern world shows that losing access to critical systems may have severe consequences for businesses, infrastructures, and society at large. In the upcoming chapters, this dissertation will examine (some small parts of) these issues in detail, identifying potential threats to availability and security and proposing practical solutions to build stronger and more dependable systems.

1.1 Dissertation Outline

Following the introduction in Chapter 1, the rest of this dissertation is structured as follows:

- **Chapter 2** provides an overview of the three key research domains explored in this work:
  - **Section 2.1** examines IoT as both a victim and perpetrator of DoS attacks, along with recovery mechanisms. This includes (i) DoS mitigation techniques, featured in **Paper I** and **Paper II**, and (ii) an enhanced recovery scheme in **Paper III**.
  - **Section 2.2** introduces 5G network availability, focusing on NWDAF. Here, **Papers IV** and **V** address adversarial data contamination in mobility prediction.
  - **Section 2.3** covers software vulnerability detection using Code-LMs. This discussion sets the stage for **Papers VI** and **VII**.
- **Chapter 3** (*Contributions and Conclusions*) summarizes the main research outputs:
  - **Section 3.1** presents an overview of all seven included papers, describing their individual contributions and the author's role in each.
  - **Section 3.2** synthesizes key findings from the body of work, showing how the proposed techniques enhance system availability and security across IoT, 5G, and software ecosystems.

The second part of this dissertation contains the full text of the included publications. Although the formatting has been standardized for consistency, each paper is presented in its originally published (or under-submission) form.

Background

From the smart fridge that texts you when you're out of milk[1] to the industrial sensors controlling entire assembly lines, our digital ecosystem has never been more interconnected. According to recent forecasts, the number of internet-connected devices is on track to exceed 30 billion by 2030[2], making our reliance on technology grow faster than our streaming queues. Yet with great connectivity comes great vulnerability: cybercrime is projected to inflict damages amounting to $12 trillion annually[3] by 2025, a sum that could buy enough coffee to keep hackers (and everyone else) awake for years. This reality elevates cybersecurity from a purely technical matter to a critical economic priority, showing the need to ensure both data integrity and system availability.

The IoT ecosystem includes billions of devices, from smart sensors in industrial contexts [Law+24; Fis+24] to personal wearables [Tao+23; Ate+22]. Given the limited computational and battery resources of many IoT devices, achieving and maintaining robust availability can be more challenging than in traditional infrastructures. IoT devices are prone to be both victims (i.e., targeted by DoS attacks [Ron+17]) and part of the perpetrators (i.e., hijacked into botnets [Ant+17a]) of cyber attacks. Consequently, securing IoT availability has far-reaching implications not only for end-user services (like smart homes [PME19]) but also for critical infrastructures (such as energy grids [Sal+19]) that increasingly depend on IoT solutions. As some IoT services increasingly rely on mobile networks like 5G for connectivity, ensuring availability becomes even more complex.

As of the third quarter of 2024, global 5G connections surpassed two billion[4], reflecting a 48% year-over-year growth. This rapid adoption represents 5G's role as a dominant connectivity technology, with projections indicating 5.5 billion con-

[1] https://www.samsung.com/us/support/answer/ANS00049761/

[2] https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/iot-connections-outlook

[3] https://www.forrester.com/blogs/predictions-2025-cybersecurity-risk-privacy/

[4] https://www.businesswire.com/news/home/20241218716754/en/Global-5G-Connections-Hit-Two-Billion-Milestone-in-Q3-2024

nections by 2030[5]. 5G introduced ultra-low latency [Mag+24], high bandwidth [PCH16], and massive IoT connectivity [Che+21b], supporting everything from autonomous vehicles [Kak+24] to smart city services [She+22]. At the same time, 5G's complex architecture, including new components such as the NWDAF, creates expanded attack surfaces and new vulnerabilities. At this point, ensuring high availability in 5G not only safeguards consumer applications but also supports critical industrial and public safety services.

Regardless of whether an organization is rolling out IoT services or deploying 5G network components, software vulnerabilities remain a common denominator across the two. Attackers continue to exploit weaknesses in code to breach systems, steal data, or launch disruptive attacks that degrade availability. Automated vulnerability detection methods, ranging from static [Sad+18a] and dynamic [Bal99a] analysis to fuzzing [Mil+95] and the emerging Code-LMs [Che+21a], are increasingly essential for identifying and mitigating security flaws before they can be exploited. Reliable code translates directly into stronger system availability, thereby benefiting both IoT and 5G ecosystems (and beyond).

With this perspective in mind, the following sections explore each of the different research areas covered in this dissertation.

2.1 IoT Availability

IoT has transformed our interaction with technology, linking billions of devices and enabling numerous applications, from smart homes to healthcare (as mentioned earlier). As IoT systems integrate into daily life and critical infrastructures, ensuring security and reliability is crucial. Availability is a key aspect of IoT, vital for the operation and dependability of the devices.

In IoT, availability refers to the ability of the devices, services, and networks to be accessible and operational when needed, even in the face of adversarial conditions or failures [MK22]. It ensures that IoT systems can perform their intended functions without interruption or degradation, delivering the expected services to users and other dependent systems.

Despite its importance, ensuring availability in IoT systems presents significant challenges due to the unique characteristics and constraints of IoT environments. These challenges include:

1. **Resource constraints**: Many IoT devices have limited computational power, memory, and energy resources, making them vulnerable to resource exhaustion attacks and failures [MP21].

[5] https://www.gsma.com/newsroom/press-release/5g-momentum-continues-with-1-6-billion-connections-worldwide-rising-to-5-5-billion-by-2030-according-to-gsma-intelligence/

2. **Heterogeneity**: IoT systems often involve a diverse range of devices, protocols, and platforms, complicating the implementation of consistent security measures and recovery mechanisms [Noa+22].

3. **Large-scale deployments**: The sheer number of IoT devices deployed in various environments makes it difficult to manage, monitor, and protect them effectively against availability threats [Tag+24].

4. **Wireless connectivity**: IoT devices often rely on wireless networks, which are susceptible to interference, jamming, and other availability disruptions [KA24].

5. **Cyber-attacks**: IoT systems are increasingly targeted by sophisticated cyber-attacks, such as DoS, which can overwhelm devices and networks, rendering them unavailable [Kol+17; SM17; Ant+17a].

To address these challenges and ensure reliable operation in IoT systems, there is a need for improved security architectures and recovery mechanisms. These solutions must be crafted to reduce the effects of availability threats like DoS attacks and device failures while also considering the specific characteristics and limitations of IoT environments mentioned above. New methods, including collaborative defenses [RFB17; LKG13], distributed filtering [FAB12], and advanced recovery schemes [Xu+19; Hub+20], are some of the existing examples. The first part of this dissertation concentrates on creating and assessing such solutions to improve the availability and reliability of IoT devices and networks amidst various security issues.

Researchers have proposed various approaches to address the availability challenges in IoT systems. This section explores the related work in three key areas: DoS mitigation techniques, IoT botnet detection and prevention, and IoT recovery mechanisms.

2.1.1 DoS mitigation in IoT

Various strategies have been proposed to mitigate DoS attacks in the IoT environments, operating at both the *network* and *device* levels. In the literature, these are often referred to as *router-based* and *host-based* approaches, respectively.

Network-level (router-based) defenses

Network-level defenses aim to detect and filter malicious traffic before it reaches the victim's IoT devices. These techniques include traffic analysis [Liu+18b; ARW04], anomaly detection [DAF18], and packet filtering [El-+09; PL01]. For example, [DAF18] proposed an ML-based approach to detect and mitigate DoS attacks in IoT networks by analyzing traffic patterns and identifying malicious flows. Other researchers have explored the use of software-defined networking

(SDN) to enable dynamic and programmable network defenses against DoS attacks [EP23].

Device-level (host-based) defenses

Device-level defenses focus on strengthening the security of individual IoT devices to prevent them from being compromised and used in DoS attacks by a botnet controller. These techniques include secure authentication mechanisms [ACS17], access control, and resource management [Sfo+16]. For instance, [ACS17] proposed a lightweight authentication scheme for IoT devices using physically unclonable functions (PUFs) to prevent unauthorized access and mitigate DoS attacks. Other researchers have explored the use of resource management techniques, such as rate limiting and resource isolation, to prevent resource exhaustion attacks on IoT devices [Sfo+16].

SMACK: Short Message Authentication Check

One notable device-level solution is *SMACK* (*Short Message Authentication CheckK*), as described in [GTH15]. SMACK specifically tackles *battery exhaustion* (a severe Denial-of-Sleep/DoS attack) by letting an IoT device quickly verify incoming messages before performing any costly processing. In particular, SMACK attaches a short (16-bit) Message Authentication Code (MAC) near the start of each incoming message so that the device can immediately check if the message originates from a legitimate source.

If the short MAC verification fails, the device discards the message *before*, expending additional CPU cycles, memory, or network overhead. This saves battery life on constrained devices and prevents attackers from forcing the device to remain active or perform unnecessary parsing. In practice, SMACK can be integrated with the Constrained Application Protocol (CoAP) [SHB14] by placing the short MAC in CoAP's `Token` field, requiring no changes to the underlying message format.

A succinct representation of SMACK's short MAC computation (tailored for 16-bit fields) is as follows:

$$v_i \;=\; \big(m_{i0} \;+\; a \cdot m_{i1} \;+\; a^2 \cdot m_{i2}\big) \;\cdot\; b \;+\; c_i, \tag{2.1}$$

where

- $m_{i0}, m_{i1}, m_{i2} \in \mathrm{GF}(2^{16})$ are 16-bit chunks derived from the CoAP header plus a short request identifier,
- $a, b, c_i \in \mathrm{GF}(2^{16})$ are key values derived from a master key K_M, and
- v_i is the 16-bit output that serves as the short MAC included in the packet.

Because the device can reject invalid traffic at a very early stage, SMACK significantly reduces the impact of DoS attempts. Moreover, experimental results in [GTH15] show that SMACK's overhead is low in terms of memory footprint, CPU cycles, and energy usage, making it especially suitable for constrained IoT platforms that need robust, energy-efficient availability.

A hybrid collaborative approach: CLI-DOS

The *first paper* in this dissertation, *CLI-DOS*, proposes a collaborative defense mechanism that leverages the cooperation between IoT devices (hosts) and gateways (routers) to mitigate DoS attacks and preserve device availability. The key idea behind CLI-DOS is to offload the major computational burden of DoS mitigation from resource-constrained IoT devices to more powerful gateways while still relying on a short MAC check (SMACK) at the IoT device. By implementing SMACK on the device side, each incoming message undergoes a quick legitimacy check (via a 16-bit MAC) before the device engages in heavier processing. Meanwhile, the gateway uses feedback from these SMACK checks, plus additional filtering rules, to drop suspicious traffic at the network perimeter.

Through this sync, gateways analyze incoming flows and block identified attack messages before they can overwhelm the IoT devices. In essence, CLI-DOS addresses varying intensities of DoS attacks aimed at exhausting device resources: if SMACK identifies messages as invalid, the gateway quickly blocks further requests of the same type, preventing resource-strapped IoT devices from being flooded. By combining elements of both router-based (network-level) and host-based (device-level) defenses, CLI-DOS effectively mitigates DoS attacks while maintaining service availability.

2.1.2 IoT Botnet Detection and Prevention

Modern IoT ecosystems comprise a vast number of connected devices, from household appliances to large-scale industrial sensors, each potentially vulnerable to malicious activities. Among the most critical security challenges is the rise of IoT botnets, which aggregate compromised IoT devices into large-scale networks capable of launching various cyberattacks, particularly Distributed Denial-of-Service (DDoS). In recent years, attacks such as Mirai [Kol+17] shows how rapidly IoT devices could be hijacked at scale, targeting high-profile websites[6] and services[7,8]. Consequently, research and industry efforts have produced a variety of strategies to detect, block, and mitigate IoT-specific botnet threats.

However, many of these existing solutions face practical limitations when deployed at scale:

[6] https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/
[7] https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/
[8] https://www.wired.com/2016/10/internet-outage-ddos-dns-dyn/

- **IoT resource constraints.** As also mentioned in section 2.1, many IoT devices are extremely resource-constrained (in terms of CPU, memory, or battery) [MP21]. Sophisticated techniques can exceed what small IoT nodes can handle locally.

- **Centralized bottlenecks.** Traditional DDoS mitigation often focuses on core networks or cloud layers [FAB12], which can become overwhelmed [Sha+15b] under large-scale attacks targeting IoT ecosystems.

- **Insufficient source-based controls.** Without enforcing traffic filtering near the source [MR05], attackers can re-route or spoof malicious traffic, circumventing defenses.

These gaps show the need for lightweight *source-based* mechanisms capable of rapidly filtering malicious traffic close to its origin. The fact that (i) IoT devices do not typically connect to the general internet but rather specific to the particular backend(s), and (ii) IoT devices that are infected by botnet are already undesirable from the resource perspective gives the device owner incentives to implement DDoS mitigation technique at the source (not just on the network).

The remainder of this section surveys key detection and mitigation paradigms.

Signature-Based and Anomaly-Based Detection Methods

Broadly, detection mechanisms for IoT botnets draw on two dominant paradigms: signature-based and anomaly-based.

- **Signature-Based Methods** rely on predefined rules or patterns (*signatures*) characteristic of known botnets to identify and block malicious traffic [GSG19; KL20]. Classical IDS tools such as Snort[9] [Cas+03] or Suricata[10] rely heavily on such signatures for fast and accurate detection of repeated, well-understood attacks. However, these approaches are less effective for novel threats whose behaviors have not yet been profiled or whose signatures rapidly evolve.

- **Anomaly-Based Methods** aim to detect deviations from normal IoT device behavior [Kom+21; Bor+23; NB18]. For instance, [Mei+18] proposed an ML-driven anomaly detection system that establishes baseline profiles of device traffic and flags unusual patterns. This technique is better suited for unknown or zero-day threats, as it does not depend on prior knowledge of an attack signature. Nonetheless, anomaly-based solutions often require periodic retraining, careful tuning to minimize false positives, and sufficient compute resources for full traffic visibility.

[9] https://www.snort.org/
[10] https://suricata.io/

Both methods can be complementary: signature-based filters can quickly block well-known exploits, while anomaly-based models catch stealthier or evolving attacks.

Collaborative and Distributed Mitigation Strategies

Beyond detection, mitigation of IoT botnets increasingly leverages collaborative or distributed strategies. This involves cooperation among devices, gateways, Internet Service Providers (ISPs), and cloud services to coordinate a faster, more scalable response.

FireCol [FAB12], for example, detects DDoS attacks at the ISP level and disseminates alerts upstream. If FireCol identifies an attack in one ISP domain, it triggers a subscription-based mechanism that notifies other ISPs, which then adopt similar mitigation measures. Such collaboration effectively handles high-volume attacks by filtering out malicious traffic as early as possible in the network path.

Similarly, multi-access edge computing (MEC) [Kek+18] can be harnessed to do detection or throttling logic closer to the sources of IoT traffic. By implementing filtering at the network edge, i.e., near a cellular base station or local gateway, the approach can block malicious packets before they converge on a single target, reducing both bandwidth consumption and the chance of overwhelming the victim. These ideas naturally align with *source-based* defenses such as D-WARD [MR05], which perform traffic policing near the origin to contain attacks early.

Nevertheless, these strategies can still suffer from deployment complexity, overhead in handling high data rates, or slow synchronization among cooperative nodes. For resource-starved IoT endpoints, relying purely on device-side detection is often impractical, while large centralized systems can quickly become overloaded during a volumetric DDoS.

From Collaborative Approaches to X-Pro

Building on collaborative, edge-based filtering concepts, the *second paper*, **X-Pro**, introduces a distributed proxy architecture for IoT-specific DDoS mitigation. At its core, X-Pro leverages the *eXpress Data Path* (XDP) for real-time, in-kernel packet filtering at line speed, allowing malicious flows to be blocked before they reach core networks. Multiple X-Pro proxies are deployed near traffic sources, i.e., ISP points of presence (PoP), edge gateways, or cloud backends, and share aggregated flow metrics via a lightweight synchronization mechanism through a centralized database. This setup detects both massive volumetric floods and subtler multi-proxy attacks by imposing threshold-based controls on IoT traffic at the source. Consequently, X-Pro balances minimal overhead with robust source-end protection and can integrate into diverse IoT environments, including 5G MEC nodes or on-premise gateways.

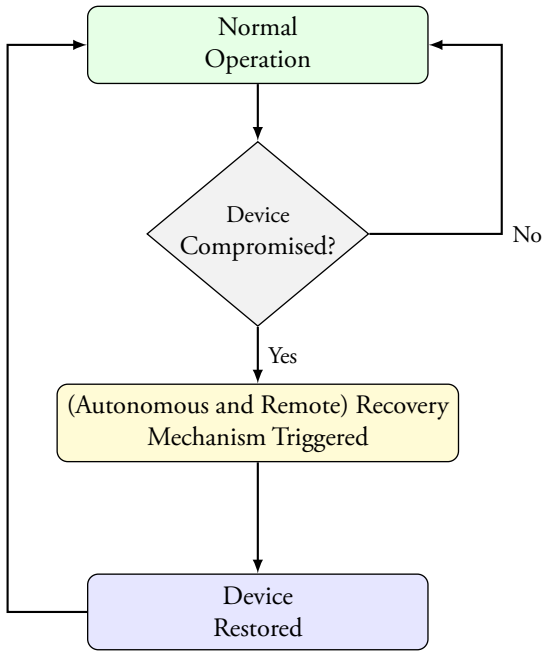

Figure 2.1: High-level flow of IoT recovery.

2.1.3 IoT Recovery

IoT device recovery refers to the process of restoring a device to a secure and operational state after it has been compromised by malware, software corruption, or hardware attacks [Xu+19]. Since many IoT devices operate in unattended or resource-constrained environments, ensuring reliable recovery is a significant challenge. Unlike traditional computing systems, where recovery may involve user intervention or reinstallation, IoT recovery mechanisms often need to function autonomously and remotely. A high-level flow of a recovery in IoT is depicted in figure 2.1.

Various researchers have explored mechanisms to enhance the recovery and resilience of IoT systems. These approaches can be broadly categorized into two main strategies: remote recovery and self-healing.

Remote Recovery

Remote recovery focuses on restoring compromised devices through the intervention of an external entity, such as a remote administrator or a trusted server. Notable examples include:

- **APEX** [Nun+20] offers a system that leverages a Trusted Execution Environment (TEE) to allow a remote administrator to recover an embedded device

even when parts of the software stack are compromised. It also exposes an interface to intrusion detection software, enabling automated recovery triggers.

- **RO-IoT** [Suz+20] autonomously reboots and reinstalls a fresh OS image from a trusted server if the existing one is compromised, ensuring the authenticity and integrity of the recovery process.
- **SeED** [ISZ17] is a non-interactive attestation protocol where devices initiate attestation at random intervals, eliminating the need for verifier challenges and reducing communication overhead. It uses a Pseudorandom Number Generator (PRNG) to randomize attestation timing. SeED relies on hardware-assisted protections to safeguard attestation keys, software integrity, and timestamps.

Self-Healing

Self-healing approaches focus on enabling IoT devices to detect and recover from compromises independently, without relying on external intervention. Key examples include:

- **Verify&Revive** [AC20] combines remote attestation with a mechanism to restore software to a known-good state if the device is found compromised. It uses Merkle hash trees [Mer88] to pinpoint corrupted memory regions.
- **HEALED** [IST19] similarly integrates attestation with a disinfection protocol that resets corrupted software. Its Merkle tree structure provides efficient modification tracking.

Countering Physical Attacks

In addition to the above strategies, **CASU** [De +22] is a system architecture designed to counter physical attacks that wear out flash memory, potentially leading to hardware destruction. CASU provides secure update and recovery capabilities through a TEE, addressing both software and hardware threats.

CIDER and Lazarus

Among these IoT recovery solutions, *CIDER* [Xu+19] and *Lazarus* [Hub+20] stand out for their promise of *unconditional recovery* even if a device's firmware has been completely compromised. In contrast to earlier recovery mechanisms, where success may rely on some part of the local firmware or an intact recovery partition, CIDER and Lazarus both employ hardware or TEE-enforced watchdog logic to forcibly reset and verify each attempted boot. As a result, adversaries cannot indefinitely prevent a return to known-good firmware by subverting local code. CIDER introduces the notion of "dominance," meaning that within a bounded timeframe, the device owner or administrator can remotely enforce the

execution of specific firmware on an IoT device, regardless of the device's current compromised state. This bounded timeframe is closely related to the concept of the Authenticated Watchdog Timer (AWDT), which ensures resets cannot be locally bypassed by malicious firmware.

Authenticated Watchdog Timer (AWDT). An AWDT functions like a conventional watchdog timer in that it resets the system if not periodically serviced, but with a critical security enhancement: it requires cryptographically protected keepalive messages (called "DeferralTickets") issued by a remote administrator to defer the platform reset. This means that, unlike a conventional watchdog timer that can be serviced independently by local firmware, an AWDT cannot be serviced by potentially compromised local firmware without authorization from the trusted remote hub.

CIDER Overview. CIDER achieves dominance via two core components:

1. *Gated Boot*: Only authorized firmware is allowed to execute on the device.
2. *Reset Trigger*: A trusted hub can force the device to reset, invoke the gated boot procedure, and regain full control of a compromised device.

This design centers on simple hardware primitives, including latches and an AWDT. CIDER thus ensures that if an attacker disrupts normal device execution, the watchdog and reset trigger can revert the device to a secure state. However, note that AWDT implementation on CIDER requires an additional MCU to be attached to the main board.

Lazarus Overview. Building on CIDER's fundamentals, *Lazarus* aims to make unconditional recovery *practical* for low-end microcontrollers. Instead of adding new hardware like CIDER, Lazarus uses the TEE capabilities of modern MCU platforms (i.e., ARM TrustZone) to implement the necessary security primitives (i.e., the AWDT, storage latches) in software.

During boot, Lazarus's minimal trusted computing base (TCB) configures the TEE so that critical peripherals and data remain protected from untrusted firmware. A component called *TEETrigger* enforces resets if the device fails to receive fresh, authenticated tickets from the hub, ensuring reversion to a known-good state.

To enable seamless updates of the TCB while preserving device identity, Lazarus leverages and extends the Device Identifier Composition Engine (DICE) standard [Gro21; JPF17; Tao+21]. DICE establishes a unique device identity and a chain of trust rooted in hardware. It uses a unique secret per device, the Unique Device Secret (UDS), to derive a Compound Device Identifier (CDI) through cryptographic operations. The CDI is then used to generate a DeviceID key pair for device authentication.

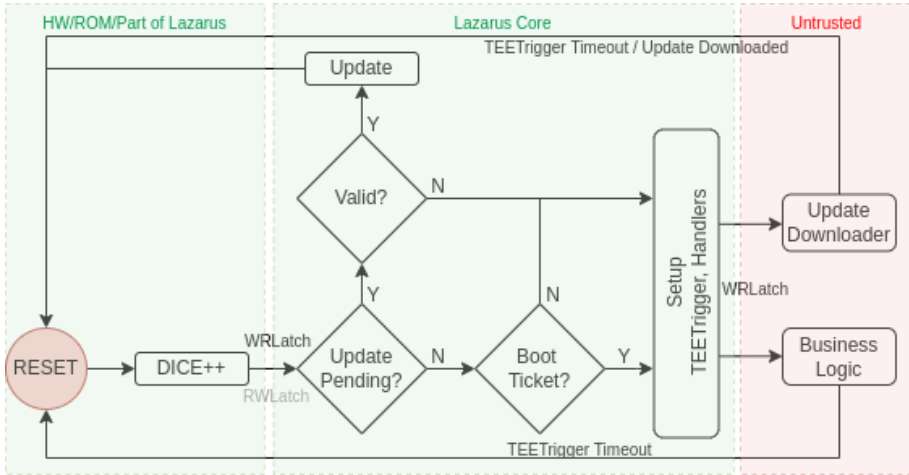

Figure 2.2: Boot flow of Lazarus.

However, DICE alone does not support firmware updates, as changes to the firmware would alter the CDI and DeviceID. To address this, Lazarus introduces DICE++, an extension to DICE. DICE++ adds a static secret `static_sym` derived from the UDS and a static device identifier `dev_uuid` generated at first boot. By combining `static_sym` with a measurement (hash) of the current firmware version, Lazarus can generate a firmware-specific authentication key, `core_auth`. In other words, `core_auth` is *bound to the current firmware version* because it changes whenever the firmware image is updated; if an attacker modifies the firmware or tries to reuse an old `core_auth` on new code, the cryptographic derivation will not match.

After a firmware update, Lazarus uses the new `core_auth` to produce an "identity token" `dev_auth` that proves to the hub that the update was applied on the same physical device hardware. Here, *same device hardware* means that both the old and new firmware derive their keys from the same underlying UDS and `dev_uuid`. As a result, the hub can confirm that it is still talking to the exact same physical chip despite a firmware update that would ordinarily change the CDI and DeviceID. This allows the hub to link the device's new DeviceID (resulting from the firmware change) to its previous identity, ensuring that updates do not break existing device identity records.

As illustrated in Figure 2.2, the Lazarus boot flow implements a secure chain of execution. After reset, DICE++ runs in secure privileged mode, configuring protections and handling the cryptographic derivations for identity preservation. Lazarus Core then executes in secure unprivileged mode, verifying and applying any pending updates or obtaining boot tickets from the hub via the Update Downloader (which runs in the non-secure world). Before transferring control to untrusted software, Lazarus Core initializes TEETrigger and configures handlers for

critical peripherals, ensuring these protections remain active during normal operation.

Both CIDER and Lazarus rely on remote attestation, secure boot, and authenticated tickets (`DeferralTickets` and `BootTickets`) to maintain device integrity and recovery capabilities. Their threat models assume that adversaries **can** eavesdrop on, tamper with, or temporarily block communications with the trusted hub but cannot do so indefinitely.

Enhancements Proposed in the Third Paper

While unconditional recovery solutions like CIDER and Lazarus guarantee that a remote owner can eventually enforce known-valid firmware, they can get stuck in perpetual reset when under persistent network tampering or runtime attacks that sabotage ticket retrieval. In such scenarios, the AWDT forces endless reboots because the device can neither receive valid tickets nor differentiate an external (network-based) cause from an internal (runtime) one.

To address these limitations, our third paper adds a *verification mode* to diagnose suspected disruptions, enabling the device to switch to a *minimal application mode* if external connectivity remains blocked. This prevents lockout and maintains critical functionality while awaiting restored connectivity or hub commands. Additionally, embedding "action commands" into ticket messages helps the device selectively respond (i.e., apply a firmware update, re-attest) rather than blindly resetting on every missed ticket. As a result, these enhancements preserve availability under worst-case network conditions and runtime exploits, ensuring more resilient "best-effort dominance."

2.2 5G Availability

2.2.1 NWDAF in 5G Networks

5G networks have introduced various concepts and architectural elements to handle increasingly complex services. Among these elements, NWDAF plays an important role in supporting data-driven decision-making. 3GPP [3GP22c; 3GP22a] defines NWDAF as a network function (NF) that collects, processes, and analyzes data from various sources across the 5G core network.

Role and Importance in 5G. As can be seen in figure 2.3, the role of NWDAF in 5G is rooted in its ability to aggregate data from multiple entities such as the Access and Mobility Management Function (AMF) [3GP24b], Session Management Function (SMF) [3GP24a], Policy Control Function (PCF) [3GP24c], and others. NWDAF provides the intelligence layer to optimize end-to-end service delivery by collecting statistics on various data sources. In effect, NWDAF helps

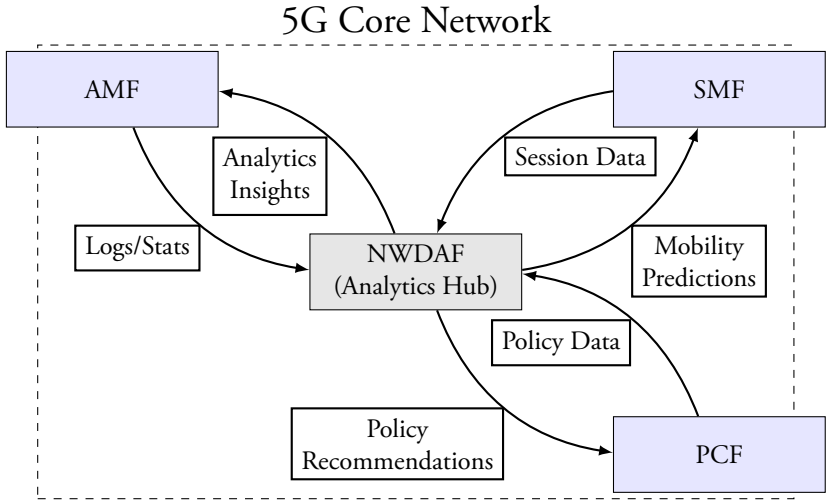

Figure 2.3: High-level overview of NWDAF in the 5G Core, illustrating data flows and analytics outputs to core functions such as AMF, SMF, and PCF.

operators identify bottlenecks, predict network loads, and proactively allocate resources to ensure a reliable and resilient infrastructure[11].

Key Functionalities and Use Cases. NWDAF's design principles revolve around leveraging ML and statistical analysis to enable a variety of analytic services [3GP22c; 3GP22a; Yua+22], including:

- **Network Performance Analytics:** NWDAF gathers and correlates performance metrics (i.e., throughput, latency, packet loss) to detect anomalies and trending issues. This allows operators to perform root cause analysis and preemptively address performance degradation.
- **Subscriber Behavior Analytics:** By examining user activity patterns, such as mobility, app usage, and QoS requirements, NWDAF can help tailor personalized policies. This extends to optimizing subscriber plans, enforcing dynamic data rates, and ensuring fair resource allocation.
- **Traffic Forecasting:** One of NWDAF's most prominent capabilities is the forecasting of network load and traffic patterns. Accurate forecasts facilitate capacity planning, load balancing, and energy-efficient scheduling.
- **Security and Fraud Detection:** NWDAF's real-time analytics can also be applied to detect suspicious activities, identify fraud, and enforce network-wide security policies.

[11] https://free5gc.org/blog/20241127/20241127/

Through these functionalities, NWDAF serves as an analytics layer that informs and orchestrates the decisions of other 5G core components.

Mobility Prediction as a Critical Service. Among the numerous applications of NWDAF, mobility prediction has gained particular attention due to the increasingly dynamic nature of 5G services. With subscribers frequently switching between cells and exploiting different access technologies (i.e., 5G gNodeB, LTE eNodeB, Wi-Fi), accurate mobility forecasts become vital. NWDAF leverages historical mobility traces and contextual information, such as user device capabilities, geographical location, and signal strength, to project future mobility patterns. These mobility insights can then be used to:

- **Optimize Handover Decisions:** By predicting a user's next cell, the network can proactively allocate resources and avoid dropped calls or service disruptions.
- **Enable Edge-Resource Allocation:** Edge computing platforms can preemptively cache or allocate computational resources near the user's predicted location, minimizing latency.
- **Improve Energy Efficiency:** Since the operator knows the movement in advance, resources that are not needed can be kept idle, saving the power.

Successful mobility prediction not only ensures a consistent user experience but also drives better utilization of network resources.

2.2.2 Mobility Prediction Fundamentals

Mobility prediction plays a pivotal role in 5G, where the user base is large and highly diverse in terms of mobility behavior [Jeo+21a]. This section outlines the core concepts of mobility prediction.

Basic Concepts of Mobility Prediction

In the broadest sense, mobility prediction refers to forecasting a User Equipment (UE)'s future location and the time it will remain connected to a particular cell or network node [ZD19]. As UE continuously moves, especially in dense urban environments, accurate predictions are challenging yet increasingly critical. From a high-level perspective (also shown in a pipeline format in figure 2.4), mobility prediction often involves the following steps:

- Data Collection: Collect location-related features, i.e., timestamps, cell IDs (or eNodeB/gNodeB IDs), signal strengths, and contextual factors such as time-of-day or historical mobility patterns.

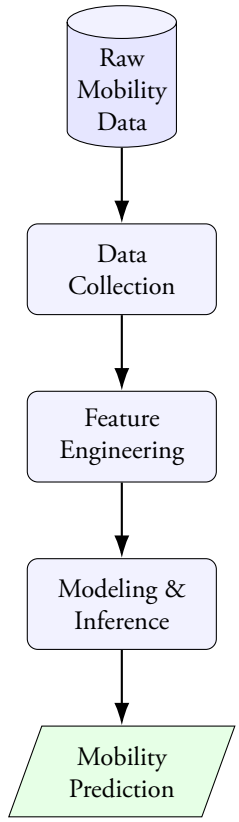

Figure 2.4: A simplified pipeline for mobility prediction, from raw mobility data to final predicted trajectories.

- Feature Engineering: Transform raw mobility data (i.e., time-series of locations) into a structured input (i.e., previous cell attachments, connection durations, signal strength distributions) suitable for machine-learning or statistical models.
- Modeling and Inference: Apply algorithms (ranging from traditional machine-learning classifiers to deep neural networks or ensemble methods) that map the input features to a predicted future location (the where) and the duration of stay (the how long).

Based on the operator's needs, these steps can be refined and re-applied periodically or continuously to capture evolving patterns in user mobility.

Metrics for Evaluating Prediction Accuracy

Numerous metrics are employed in mobility prediction research, ranging from distance-based (i.e., Euclidean [SB14] or Haversine [Rob57] distance) to classification-oriented (i.e., accuracy, F1-score) approaches [Son+10; GHB08]. Because this dissertation focuses on discrete cell-level predictions, we adopt the classification approach with two key metrics:

- Location Accuracy $\hat{l}$. Determines how often the predicted cell (i.e., eNodeB/gNodeB) matches the actual cell to which the UE attaches at a given time.
- Timeslot Accuracy $\hat{s}$. Reflects how well the model forecasts the duration (or timeslot) that a UE stays connected to a particular cell. $\hat{s}$ is evaluated *conditioned on* $\hat{l}$ being correct.

Combining these metrics results in an overall mobility prediction accuracy:

$$\textbf{Accuracy} = \frac{\sum_{e=0}^{n}(\hat{s}_{correct}|\hat{l}_{correct})}{n} \tag{2.2}$$

which captures both correct location predictions *and* correct timeslot duration estimates divided by the total number of events e from 0 to n, where n is the total number of events for any given timestamps in the dataset. Other performance indicators, such as false handover rates, latency, or computational overhead, may also be considered depending on the operational setting and the specific goals (i.e., resource-saving vs. quality-of-experience). We will use equation 2.2 throughout the discussion for the rest of this dissertation.

With this as a basis, the dissertation explores how one could reduce the accuracy of NWDAF's prediction, mainly during training and inference.

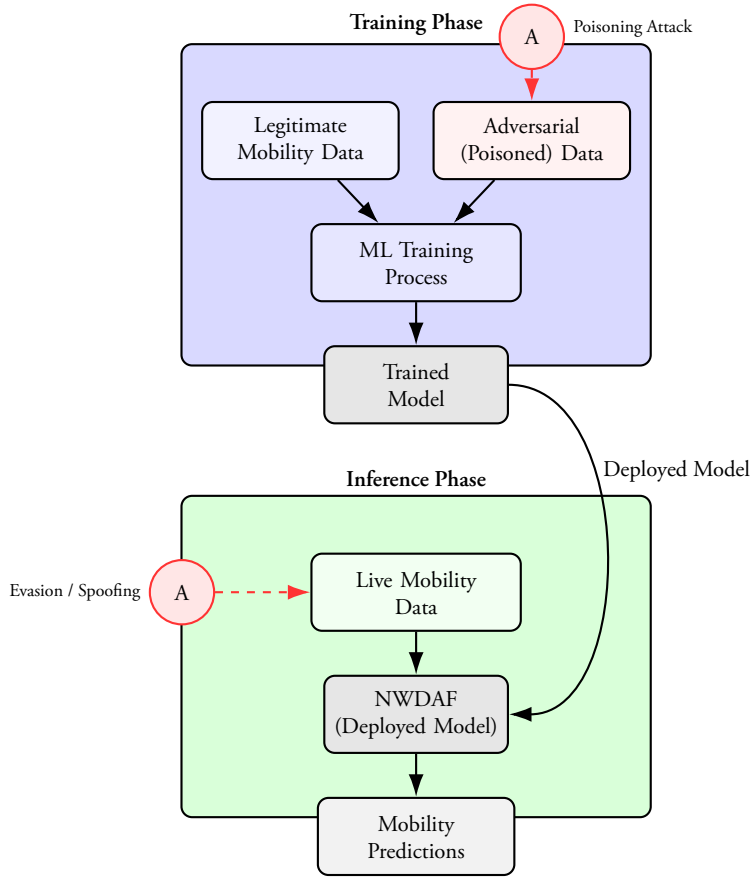

Figure 2.5: Overview of attack surfaces for NWDAF mobility prediction. During training, adversaries can inject poisoned data. During inference, they can spoof or manipulate real-time mobility data.

2.2.3 Attack Surface in NWDAF Mobility Prediction

ML models in NWDAF face security challenges that can compromise their effectiveness. The particular interest of this dissertation is looking into the challenges that arise in two main categories [OV23]: inference-time attacks and training-time attacks. Attacks in these two phases are depicted in figure 2.5.

Inference-time Attacks

Attacks at inference time involve adversarial manipulation of spatio-temporal data to degrade mobility prediction accuracy. For example, adversarial UE can inject *spoofed location updates* [Gao+23] to mislead predictive models used for handover optimization and resource allocation. Such adversarial behaviors have been studied

in the context of adversarial ML in 5G networks, where inference-time attacks, such as trajectory tampering and mobility obfuscation, can significantly reduce the effectiveness of predictive analytics [SES21].

Recent research on over-the-air (OTA) attacks demonstrates that even subtle perturbations in wireless signal-based fingerprinting systems can cause significant misclassification [Xia+23]. Given the reliance of 3GPP NWDAF on massive mobility data streams, even minor perturbations in reported positions can compound into substantial QoS degradation.

Training-time Attacks

In many NWDAF implementations, ML models are periodically retrained with newly collected mobility data. If an adversary injects malicious UE or falsified records into this training set, the resulting "poisoned" data can corrupt the learning process. Identifying and filtering these adversarial traces is often difficult, especially if the anomalies blend well with legitimate mobility patterns. Poisoned data can significantly alter the decision boundaries of ML models, leading to inaccurate predictions even after deployment. In worst-case scenarios, operators may repeatedly retrain the model on tainted data without ever regaining stable prediction performance. For instance, a study on poisoning attacks against deep learning-based wireless traffic prediction demonstrated that adversaries could manipulate training data to degrade prediction accuracy [ZL22].

In the 4^{th} paper of this dissertation, we investigated potential attacks against the NWDAF's mobility prediction models during the inference phase. We demonstrated that an adversary with the ability to hijack and clone cellular devices can significantly reduce prediction accuracy from 75% to 40% using just 100 adversarial UEs out of 10,000 total subscribers. While the specific defense mechanism depends on the attack type and mobility patterns in a particular area, we showed that basic KMeans clustering is able to distinguish legitimate and adversarial UEs (at least in our dataset).

Building upon these findings, the 5^{th} paper explored the impact of the same attacks of the 4^{th} paper on NWDAF mobility prediction models during the retraining phase, when the models are updated with new data that adversaries may poison. We evaluated different strategies for employing AutoML to maintain prediction accuracy in the face of such attacks. Our results showed that extensively tuning the initial model using AutoML and then retraining that selected model on new data, even if poisoned, outperforms reselecting models with AutoML at each retraining step. This suggests that mobile network operators should prioritize initial model selection and tuning to ensure robustness against potential data poisoning attacks during future retraining.

2.3 Vulnerability Detection

Modern software development faces a critical challenge: maintaining security while delivering frequent updates. Crowdstrike 2024 state of application security report[12] states that "*71% of organizations report releasing application updates at least once a week*". Considering this, ensuring that every release is secure is crucial. The simplest way is to check the code manually [BB13] before every release; however, as the codebase grows, this approach becomes less effective. The time and effort required to thoroughly review each line of code increases, making manual review less efficient. Additionally, people might make mistakes for different reasons; they may be tired or have differing levels of expertise, leading to inconsistent assessments.

A better approach is to employ automated tools to detect potential vulnerabilities in the codebase. These tools can be used to assist manual code review or as standalone, autonomous systems. They analyze the code systematically and consistently, flagging areas of concern for further manual review when used in conjunction with human reviewers. When used autonomously, the effectiveness of the security assessment depends on the specific tool and its capabilities. In either case, these tools allow development teams to focus their limited time and attention on the highest-risk areas.

To gain a deeper insight into the evolution of vulnerability detection systems and their significance in the software development landscape, it is essential to reflect on the historical development of these tools over time. This context will provide a more comprehensive understanding of the background of this dissertation.

2.3.1 Traditional Approaches

Traditional vulnerability detection approaches include *manual code review*, *static analysis*, and *dynamic analysis*. A summary and the relationship among these three are depicted in figure 2.6.

Manual Code Review

Manual code review involves security experts or developers meticulously inspecting source code to identify potential vulnerabilities [Sad+18b; BB22]. The effectiveness of this method depends on the reviewers' expertise and familiarity with both the codebase and secure coding practices. Although thorough code reviews can uncover subtle logic flaws and design issues, this process is time-consuming and does not scale well for large codebases [LE20]. Moreover, human error and reviewer fatigue can lead to missed weaknesses, particularly if the codebase is complex or if deadlines are tight.

[12] https://www.crowdstrike.com/en-us/2024-state-of-application-security-report/

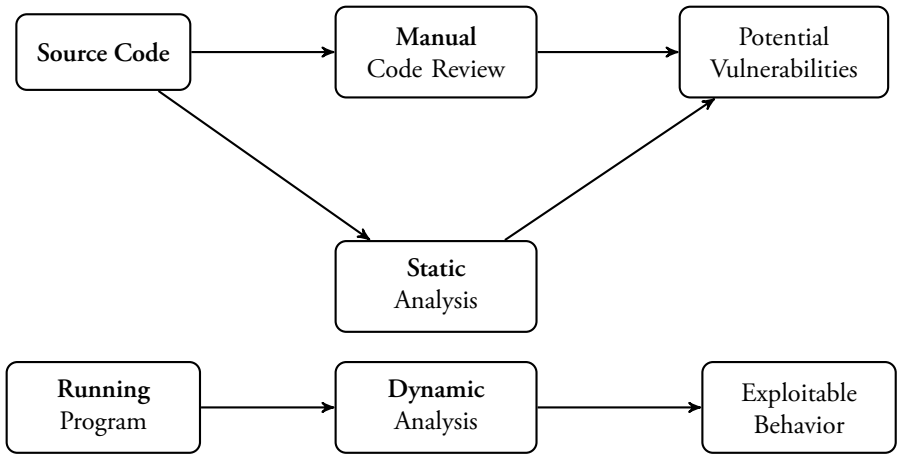

Figure 2.6: A high-level comparison of three traditional approaches to vulnerability detection. **Manual Code Review** relies on human inspection of source code, **Static Analysis** inspects code without running it, and **Dynamic Analysis** examines a running program.

Static Analysis

The static analysis aims to detect vulnerabilities by examining source code or bytecode without executing it [Tho21; CM04]. Popular static analysis tools [13,14] scan for known bug patterns, such as buffer overflows [FOB05], injection points [LL05], and unsafe function calls[15]. These tools can handle large codebases faster than manual inspection and can integrate into continuous integration (CI) pipelines, thereby providing automated checks early in the development cycle.

However, a known drawback of static analysis techniques is their often high rate of false positives [Tym17]. Developers can become overwhelmed by the volume of warnings, many of which may not be relevant. Moreover, static analysis relies on *explicit rule sets* or pattern-matching heuristics. While these rules are typically informed by known vulnerabilities and established coding standards (i.e., CWE[16] and MISRA[17]), they must be continuously updated to address *newly discovered exploits* or subtle variations of existing ones. This maintenance overhead can be tedious and costly, especially in large organizations where codebases and threats rapidly evolve.

Another limitation is that static analysis may miss vulnerabilities that depend on runtime information, such as environment-specific configurations, dynamic li-

[13] https://semgrep.dev/products/semgrep-code/
[14] https://www.sonarsource.com/products/sonarqube/
[15] https://cwe.mitre.org/data/definitions/676.html
[16] Common Weakness Enumeration; https://cwe.mitre.org/
[17] Motor Industry Software Reliability Association; https://misra.org.uk/

brary loading, and input-dependent behavior. Since code is not actually executed, these context-dependent issues are often undetectable through static analysis alone. As a result, static analysis tools are frequently complemented by other methods such as dynamic analysis, manual review, or more advanced AI-based techniques to improve both coverage and accuracy.

Dynamic Analysis

Dynamic analysis inspects a program's behavior during execution [Bal99b; SAB10]. Typical techniques include instrumenting applications to monitor memory usage, execution paths, and system calls under various test inputs. Examples range from unit tests and integration tests to more specialized methods such as fuzzing [MW23; Fio+20; Ngu+20], where random or semi-random inputs are systematically generated to trigger unexpected states. Dynamic approaches can catch vulnerabilities (i.e., memory corruption or resource leaks) that are difficult to detect statically. However, they inherently rely on the coverage and quality of the test inputs. Vulnerabilities that are not triggered by a given test set remain undiscovered, limiting the completeness of dynamic analysis. In addition, dynamic methods can incur significant overhead in terms of execution time and resource usage when dealing with large-scale systems.

Limitations and Motivations for Advanced Techniques

While these traditional approaches have proven effective for certain categories of vulnerabilities, each suffers from shortcomings that motivate the pursuit of more advanced and automated methods. Manual code reviews are labor-intensive and dependent on expert knowledge, static analysis may generate a substantial number of false positives, and dynamic analysis can miss vulnerabilities not triggered by specific inputs. Consequently, recent research has moved toward more sophisticated techniques, such as hybrid analysis (combining static and dynamic methods) and ML-based vulnerability detection, to address these gaps. These emerging solutions seek to reduce manual effort, improve accuracy, and increase the coverage of potential security threats beyond the capabilities of traditional methods [Tho21; Tym17].

2.3.2 Early Neural Network Approaches for Vulnerability Detection

As deep learning gained traction in fields such as computer vision and natural language processing, researchers began to explore its potential for security-related tasks. In recent years, various deep learning-based methods have been proposed for vulnerability detection and related software engineering tasks [Cha+22b; Li+18a; Zho+19a]. These approaches capitalize on the capacity of neural networks to automatically learn features from large-scale datasets, reducing the burden of handcrafting features. These techniques generally focus on traditional deep learning

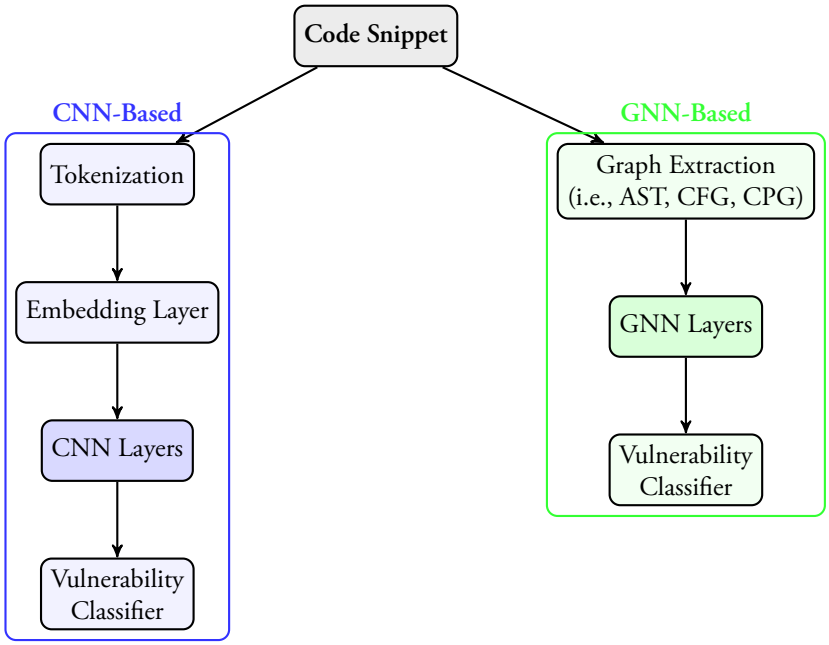

Figure 2.7: Side-by-side illustration of two early deep learning approaches to vulnerability detection. **Left: CNN-based** methods process token sequences, often with a token embedding layer feeding into convolutional filters. **Right: GNN-based** methods construct a graph (i.e., abstract syntax tree and code property graph) and then use graph neural networks to propagate information and classify potential vulnerabilities.

backbones such as convolutional neural networks (CNNs) [LeC+89] and graph neural networks (GNNs) [Sca+09]. The summary is depicted in figure 2.7.

CNN-based Methods

One of the representative CNN-based approaches is *VulDeePecker* [Li+18a], which detects vulnerabilities in C/C++ programs by learning features from slices of code. CNNs are adept at capturing local patterns in their input data; thus, when applied to code, CNN filters learn localized features such as syntax elements or short-range data dependencies. This approach can yield promising detection rates and reduce the need for extensive manual feature engineering. However, CNNs have inherent limitations in capturing long-range dependencies and context spread over multiple files or functions, which are often crucial in identifying certain types of software vulnerabilities.

GNN-based Methods

Graph neural networks leverage graph representations of source code, such as abstract syntax trees (ASTs), control flow graphs (CFGs), or code property graphs (CPGs). *Devign* [Zho+19a] is an example of this category by combining GNNs with CPGs to detect vulnerabilities in C/C++ programs. GNNs are uniquely positioned to capture structural and relational information in code, such as function calls, variable usages, and data-flow links. By encoding this information into a graph structure, GNNs can learn context-specific representations that go beyond token-level patterns. Despite these strengths, GNNs can still face challenges in scaling to large codebases and effectively modeling highly complex control flows, particularly when the number of nodes and edges grows significantly.

Limitations of Early Neural Network Approaches

While CNN- and GNN-based models have shown promising results in vulnerability detection, there remain notable drawbacks. One key limitation is the need for substantial feature engineering. Although these deep learning models automate part of the feature-learning process, researchers often manually design code representations (i.e., ASTs or CPGs) before training the neural network [Yan+22]. This step can be time-consuming, requiring deep domain expertise to capture the nuances of different programming languages and vulnerability types. Moreover, traditional neural architectures may struggle with the complex semantics and control-flow relationships inherent to source code. Long-range dependencies, such as those spanning multiple function calls or files, are especially difficult to model with CNNs, whereas GNNs may be constrained by the complexity of graph construction.

Why Language Models?

Recent advances in transformer-based architectures and attention mechanisms offer a potential solution to these challenges by capturing long-range dependencies and global context within code. Such mechanisms have become integral to language model (LM)-based methods, which are often pre-trained on massive repositories of source code from diverse languages and projects. This extensive pre-training enables language models to learn rich contextual embeddings and semantic relationships within code, potentially bridging some of the gaps left by prior CNN- and GNN-based methods.

2.3.3 Language Models

Language models (LMs) represent a class of deep learning approaches trained on vast amounts of text data to learn nuanced, context-aware representations of natural language. These models, such as BERT [Dev+19a], GPT [Rad+19], Gemini

[Tea24], and Deepseek [Dai+24], have achieved state-of-the-art (SOTA) performance on a wide array of NLP tasks including question answering, language translation, and text generation.

A key factor in the success of modern LMs is the *transformer* architecture [Vas+17], which employs self-attention mechanisms to capture long-range dependencies in text. This design excels at modeling contextual relationships, allowing the network to weigh different parts of the input sequence based on their relevance to each position. Through large-scale pre-training on immense text corpora, LMs acquire an understanding of linguistic structures (i.e., syntax, semantics) and pragmatic elements (i.e., context, idiomatic usage). The result is a model capable of generating coherent, contextually nuanced text that closely resembles human writing.

Given their robust language-understanding capabilities, there has been growing interest in applying LMs to other domains, particularly *source code* analysis and generation. Codebase shares structural similarities with natural language (i.e., syntax and grammar), yet it also presents unique challenges such as strict compilation requirements, domain-specific libraries, and varying coding styles. Despite these challenges, LMs have been adapted for tasks like *code completion* [Svy+20], where the model predicts the next token or line of code based on context, and *code summarization* [Fen+20a], where the goal is to generate concise, human-readable explanations of code snippets. Furthermore, researchers have also investigated *bug detection* [Li+24] tasks, leveraging LMs to identify syntax anomalies or suspicious programming patterns.

Overall, the integration of LMs from natural language tasks into software engineering has shown promising results, particularly due to their ability to learn representations that capture both syntactic structures and higher-level semantics. As these models continue to scale in size and training data, their capacity to handle sophisticated code analysis tasks, including vulnerability detection, stands to grow.

LM for Code Analysis

Source code, much like natural language, contains rich structural and semantic characteristics that can be modeled with modern deep-learning approaches. LMs have shown promise in a wide range of code analysis tasks by leveraging their ability to capture complex patterns and contextual relationships in textual data [Alo+19; Nam+24; Shi+24].

Over the past few years, researchers have developed specialized LMs tailored to source code. One prominent example is *CodeBERT* [Fen+20a], a model pre-trained on a massive corpus of programming language data (paired with corresponding natural language descriptions). By learning from these parallel data sources, CodeBERT is able to capture both the syntactic structure and high-level

semantics of code, which can aid in downstream tasks such as bug detection and code search.

Building on CodeBERT's success, *GraphCodeBERT* [Guo+21] incorporates *graph-based* representations of code to model information contained in abstract syntax trees (ASTs) and data-flow graphs (DFGs). This graph-enhanced approach addresses the inherent structural properties of source code more directly by representing the relationships between variables, methods, and control-flow nodes. Empirical results show that GraphCodeBERT can outperform earlier models in tasks like code summarization, code completion, and defect detection, showing the significance of capturing both token-level context (as in language) and structural properties (unique to code).

These advancements indicate that language models, particularly those refined for source code, can more effectively grasp the complex relationships in large-scale software repositories. Consequently, they offer a powerful foundation for applications such as automated vulnerability detection, where a nuanced understanding of logic flow, data dependencies, and syntactical correctness is crucial. As LMs continue to improve and incorporate richer structural and semantic insights, their application to code analysis is set to expand, potentially transforming how developers assess code quality and software security.

Recent years have witnessed growing interest in leveraging language models for vulnerability detection, particularly in C/C++ code. While studies report promising results, they also show significant technical and methodological challenges in using LMs for robust and generalizable vulnerability detection.

A critical step in enhancing the efficacy of LMs for this purpose involves understanding the underlying mechanisms of fine-tuning. Fine-tuning enables the adaptation of a pre-trained model to specific tasks, such as identifying vulnerabilities. This process not only aligns the model closer to the target domain but also improves its predictive accuracy. Through fine-tuning, we can effectively tailor the general-purpose representations learned during pre-training to detect subtle patterns indicative of vulnerabilities within the code.

Fine-Tuning Process for LM-based Vulnerability Detection. Let $f_{\boldsymbol{\theta}}$ be a large language model (i.e., a Transformer) with parameters $\boldsymbol{\theta}$ that has been *pre-trained* on large-scale text or code data. During pre-training, $\boldsymbol{\theta}$ learns general-purpose representations (i.e., by predicting masked tokens or next tokens). Figure 2.8 summarizes the fine-tuning process (in a simplified version).

To adapt this model for vulnerability detection, we attach a lightweight classifier head $h_{\boldsymbol{\phi}}$ (in practice, this attachment is implemented through standard model composition methods available in various deep learning libraries[18]), often a feed-forward network or a linear layer, on top of $f_{\boldsymbol{\theta}}$. For a given snippet of code x,

[18]One example is Hugging Face Transformers' AutoModelForSequenceClassification: https://huggingface.co/docs/transformers/main/model_doc/auto#automodelforsequenceclassification

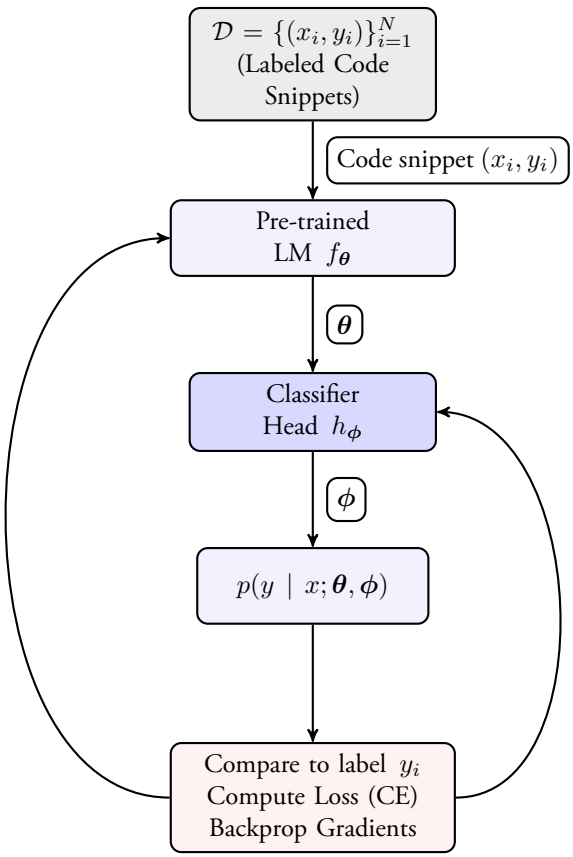

Figure 2.8: Fine-tuning a pre-trained language model $f_{\boldsymbol{\theta}}$ for vulnerability detection. Each labeled code snippet (x_i, y_i) from the dataset $\mathcal{D}$ is fed into the model and a classifier head $h_{\boldsymbol{\phi}}$. The output probability $p(y \mid x)$ is compared to the ground-truth label y_i to compute a cross-entropy (CE) loss. Gradients are then backpropagated, updating both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$.

our model outputs a probability distribution over possible labels (i.e., *vulnerable* vs. *non-vulnerable*):

$$p(y \mid x; \boldsymbol{\theta}, \boldsymbol{\phi}) \;=\; \text{softmax}\Big(h_{\boldsymbol{\phi}}\big(f_{\boldsymbol{\theta}}(x)\big)\Big),$$

where y is the binary label indicating vulnerability. We then collect a labeled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, with x_i being code snippets and $y_i \in \{0, 1\}$ the corresponding ground-truth labels. A common choice of loss function for fine-tuning is cross-entropy:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) = -\sum_{i=1}^{N} \Big[y_i \log p(y_i = 1 \mid x_i) + (1 - y_i) \log p(y_i = 0 \mid x_i) \Big].$$

Gradient-Based Optimization. To minimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$, we use gradient-based optimization, such as Adam [KB15]. The parameters are updated iteratively using gradient descent:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \, \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}),$$

$$\boldsymbol{\phi}^{(t+1)} = \boldsymbol{\phi}^{(t)} - \eta \, \nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$$

where:

- $\eta > 0$ is the *learning rate*, a hyperparameter controlling the step size of updates.
- $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$ is the gradient of the loss function with respect to $\boldsymbol{\theta}$.
- $\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$ is the gradient with respect to the classifier head parameters $\boldsymbol{\phi}$.

Parameter Initialization. Typically, the backbone model parameters $\boldsymbol{\theta}$ are initialized from pre-trained weights, allowing the model to start with substantial knowledge of code syntax and semantics. However, the classifier head parameters $\boldsymbol{\phi}$ are randomly initialized since they must be learned from scratch for the vulnerability detection task.

These steps are iterated over multiple training epochs until convergence, where the loss function no longer decreases significantly.

Why Fine-Tuning Helps. During large-scale pre-training, the model parameters $\boldsymbol{\theta}$ (i.e., the backbone of $f_{\boldsymbol{\theta}}$) learn general-purpose representations of code, capturing patterns such as function boundaries, data flow, and common library usages. Fine-tuning further adapts these representations to a security-focused task by training on the smaller, labeled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where each code snippet x_i is labeled as vulnerable ($y_i = 1$) or safe ($y_i = 0$). Specifically, the

model's output probabilities $p(y \mid x; \boldsymbol{\theta}, \boldsymbol{\phi})$ are computed by attaching and updating a lightweight classifier head $h_{\boldsymbol{\phi}}$. Through gradient-based optimization of the cross-entropy loss $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$, both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ become specialized to detect subtle vulnerability indicators (i.e., unsafe function calls). This approach reduces the need for large training sets while retaining the model's broad code understanding, making fine-tuning the standard method for repurposing pre-trained language models in software security tasks such as vulnerability detection.

Datasets and Early Findings. Two major C/C++ datasets have emerged to facilitate research in this area. *DiverseVul* [Che+23a] contains 18,945 vulnerable functions covering 150 CWEs and 330,492 non-vulnerable functions. Experiments on 11 model architectures from four families (GNN [Sca+09], RoBERTa [Liu+20], GPT-2 [Rad+19], and T5 [Wan+21c]) demonstrate that LMs typically outperform GNN-based approaches, especially when trained on larger datasets. However, the models still grapple with limited generalization capabilities when facing unseen codebases or novel vulnerabilities.

As a follow-up, *PrimeVul* [Din+25] was introduced to address known limitations of existing datasets (i.e., low label accuracy and high duplication rates). Despite containing more diverse and carefully curated examples, PrimeVul reveals that even advanced training techniques and larger LM architectures achieve subpar performance. The reported F1 scores fall short of expectations, showing that current LM solutions, while promising, are still not sufficiently robust to handle more realistic and heterogeneous vulnerability scenarios.

Key Challenges. A major shortcoming observed in both *PrimeVul* and *DiverseVul* is the low F1 scores achieved by the evaluated models. PrimeVul, in particular, shows how LMs, even those considered state-of-the-art, struggle with vulnerability detection in real-world contexts. Beyond limited generalization, several factors contribute to these challenges:

- **High Vulnerability Type Diversity.** Vulnerability categories (i.e., buffer overflows, use-after-free, integer overflows) have distinct properties, code semantics, and exploit patterns.
- **Class Imbalance.** Vulnerable functions are relatively rare compared to non-vulnerable ones ($\sim$18k vs. $\sim$330k in DiverseVul, $\sim$7k vs. $\sim$220k in PrimeVul), creating skewed distributions that can bias trained models toward predicting "safe" code [Che+23a; Din+25].
- **Inadequate Training Techniques.** Although advanced methods like contrastive learning and class weighting have been employed [Che+23a; Din+25], their impact on overall performance remains limited.

Insights from PrimeVul. Further analysis in *PrimeVul* shows three specific areas where LM-based detectors often fall short:

1. **Context Limitations.** Current approaches typically restrict the analysis to the function level, missing crucial interprocedural data flows. Identifying vulnerabilities in real-world software often requires tracing variables and control paths across multiple functions and files (an inherently more complex task).

2. **Overreliance on Textual Cues.** Models frequently base decisions on superficial textual similarities rather than understanding the core logic or root causes of vulnerabilities. This reliance can lead to missed detections or false alarms when code shows minor syntactic variations.

3. **Binary Classification Oversimplification.** Treating vulnerability detection purely as a binary classification task may miss critical nuances, such as the category or severity of the flaw. *PrimeVul* suggests that decomposing the detection pipeline into smaller reasoning steps, i.e., locating potential code hotspots, analyzing control-flow anomalies, and evaluating semantic correctness, could yield more accurate and interpretable results.

The 6^{th} paper, "Catching Common Vulnerabilities with Code Language Models," explores deeper into these challenges by investigating vulnerability detection at a more granular level. Instead of treating all vulnerabilities as a single label, we focus on classifying specific Common Weakness Enumeration (CWE) types. By training models on individual CWEs and incrementally including more types, we uncover several insights. Models trained on a single, common CWE achieve higher recall and F1 scores for that specific type, but their performance deteriorates when faced with a diverse set of vulnerabilities. However, knowledge of vulnerability in some CWEs is transferrable to other CWEs with close relationships. We also show the trade-off between specialization and generalization, i.e., expanding coverage to more CWEs can improve overall precision but at the cost of reduced per-type recall.

The 7^{th} paper, "Vulnerability Detection in Popular Programming Languages with Language Models," extends the investigation beyond C/C++ to assess LM performance on JavaScript, Java, Python, PHP, and Go. These five languages represent the top entries in the CVEFixes [BNM21a], collectively covering the majority of vulnerabilities outside C/C++. Using the CVEFixes dataset, we find that detection effectiveness varies significantly across languages. Furthermore, in-depth experiments on independent datasets show how these models generalize to external code samples, with mixed success depending on the dataset and language-specific patterns. By computing standard code complexity metrics (i.e., Halstead Effort, Cyclomatic Complexity), the paper also explores the link between complexity and detection performance. The analysis demonstrates only a weak, statistically insignificant correlation between code complexity and F1 scores.

Contributions and Conclusions

3.1 Contributions

This section provides an introduction to each contribution, details the specific contributions made by the author, and explains any modifications made to the publications to include them in this dissertation.

The following list presents the author names and their corresponding acronyms in alphabetical order: Christian Gehrmann (CG), Jakob Sternby (JS), Karim Khalil (KK), Kevin Dahlén (KD), Luis Barriga (LB), Syafiq Al Atiiq (SA), and Yachao Yuan (YY).

3.1.1 CLI-DOS: Collaborative Counteraction against Denial of Service in the Internet of Things

Content

This paper presents CLI-DOS, a collaborative mechanism between IoT gateways and devices to counteract DoS attacks. CLI-DOS combines short MAC values for attack detection with a defense strategy that offloads filtering to the gateway. The gateway filters attack messages to prevent them from reaching the IoT devices while allowing legitimate connections. The proposed solution is evaluated from several standpoints, demonstrating reduced resource utilization compared to vanilla CoAP implementations while preserving service availability even when the device is under heavy attack.

Individual Contribution

CG proposed the initial idea for this paper. SA was the developer responsible for building the proof-of-concept implementation and performing the evaluation. Additionally, SA made major contributions to the writing of the paper.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.1.2 X-Pro: Distributed XDP Proxies Against Botnets of Things

Content

X-Pro is a distributed XDP proxy solution designed to defend against DDoS attacks from IoT botnets. The proxies between the IoT devices and potential victims perform flow policing on all IoT traffic from a single administrative domain. X-Pro leverages eXpress Data Path (XDP) for efficient packet processing in the Linux kernel, i.e., the blocking happens even before the node allocates memory for the incoming packets (if the packets turn out to be malicious). The proxies work synchronously to block bogus messages and detect traffic levels exceeding predefined thresholds. Evaluations show that X-Pro efficiently blocks DoS traffic for both low-rate and massive attacks.

Individual Contribution

The idea for this paper was a collaborative effort between CG and SA. SA was the developer responsible for building the PoC implementation and conducting the evaluation. For this paper, SA took the lead and was the primary person responsible for the writing.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.1.3 Regaining Dominance in CIDER and Lazarus

Content

This paper extends the CIDER and Lazarus device recovery architectures to address two major issues: attackers making devices unavailable by manipulating tickets and runtime attacks utilizing potential weaknesses in trusted firmware. We propose a new design with richer device security state tracking, allowing devices to handle attacks more robustly and with better availability guarantees. A new boot mode, LZ_VERIFY, is introduced to verify disruption causes and enhance decision-making. Evaluations show significant improvements in device availability under both network and runtime attack conditions compared to the original Lazarus design.

Individual Contribution

CG and SA worked together to identify flaws in the CIDER and Lazarus architectures, particularly from the runtime and networking perspectives. They collaborated to develop a theoretical solution to address these issues. SA was responsible for the implementation and evaluation of the proposed solution. CG and SA contributed equally to the writing of the paper.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.1.4 Attacks Against Mobility Prediction in 5G Networks

Content

This paper investigates potential attacks against the 5G NWDAF that can significantly degrade the accuracy of mobility trajectory prediction models. Through simulations, it is shown that an adversary capable of hijacking and cloning cellular devices can reduce prediction accuracy from 75% to 40% with just 100 adversarial UEs (out of 10,000 legitimate UEs) by having cloned device pairs alternately connect to different locations to deceive the network. The attack aims to make the operator retrain prediction models with adversarial data, further harming accuracy. As a countermeasure, KMeans clustering is shown to effectively distinguish adversarial and legitimate mobility patterns (at least on our dataset), enabling filtering out malicious UEs before model retraining.

Individual Contribution

The data used in this paper was generated from a simulator provided by Ericsson, with assistance from JS and LB. SA wrote an extension for the simulator to enable it to produce adversarial mobility patterns that could be used to attack the existing legitimate mobility models. Using this extended simulator, SA performed the attack, evaluation, and analysis. SA was the main contributor to the paper's writing, with help from YY, CG, JS, and LB.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.1.5 AutoML in the Face of Adversity: Securing Mobility Predictions in NWDAF

Content

This paper investigates strategies for mobile network operators to maintain NWDAF model performance using AutoML in the face of attacks on mobility prediction. Two main retraining strategies are considered: reselecting the model using AutoML at each retraining and retaining the initial model. Under varying proportions of adversarial UEs, retraining with AutoML yields worse results than retaining a well-trained initial model selected using an extensive AutoML search during initial training. We recommend prioritizing model selection during initial training to ensure the base model is optimally tuned for subsequent retraining, even when the data for retraining is poisoned.

Individual Contribution

SA developed the initial idea and concept for this paper. YY, JS, and CG provided valuable input to refine the concept and assisted with the writing. SA was responsible for both the implementation and evaluation aspects of the work.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.1.6 Catching Common Vulnerabilities with Code Language Models

Content

This paper examines the challenges of code language model (code-LM) based vulnerability detection for C/C++, focusing on the difficulty of generalizing well across different vulnerability types. We investigate the performance of code-LMs in classifying specific vulnerability types using the PrimeVul dataset. Experiments on datasets specific to each of the most common types and cumulative datasets incorporating an increasing number of types reveal that identifying a specific vulnerability class in a dataset containing all types is challenging. However, when the task is modified to identify the most common vulnerabilities, the cumulative model outperforms previous binary classification results.

Individual Contribution

The idea and concept for this paper emerged from an ideation process involving CG and SA. SA is the main contributor to the implementation and evaluation, with the help of KD. KK helped interpret the resulting output and wrote the

initial version of the analysis. CG contributed to parts of the writing. SA was responsible for a major portion of the writing.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.1.7 Vulnerability Detection in Popular Programming Languages with Language Models

Content

This paper investigates the effectiveness of language models (LMs) for vulnerability detection in popular programming languages, including JavaScript, Java, Python, PHP, Go, and C/C++. We utilize the CVEFixes dataset to create language-specific vulnerability subsets and fine-tune state-of-the-art LMs on these subsets. The results show that vulnerability detection performance varies significantly across languages, with JavaScript showing the best performance and more practical detection capabilities than C/C++ (in terms of F1 score). The relationship between code complexity and detection performance is also examined, revealing only a weak correlation between code complexity metrics and the models' F1 scores.

Individual Contribution

The idea and concept were developed through an ideation process between CG and SA. CG contributed to some of the writing, while SA served as the lead writer for the paper. SA is mainly responsible for the implementation, model fine-tuning, and evaluation. KD assisted with the dataset preprocessing and fine-tuning.

For this Dissertation

The paper's formatting has been adjusted to maintain consistency with the overall dissertation.

3.2 Conclusions

The papers presented in this dissertation contribute to the field of cybersecurity, focusing on three main areas: IoT availability, 5G availability, and vulnerability detection. A common thread that ties these papers together is their focus on enhancing system availability in the face of various threats.

Papers I, II, and III propose novel solutions to defend against attacks that can compromise system availability in the area of IoT security. CLI-DOS (Paper I) and X-Pro (Paper II) introduce collaborative mechanisms and distributed proxy

solutions to counteract DoS and DDoS attacks. While CLI-DOS focuses on safeguarding IoT devices as potential victims, X-Pro addresses the issue of IoT devices being compromised and used as attackers in a botnet scenario; both solutions aim to maintain the availability and responsiveness of IoT devices to legitimate requests, whether they are the target or the source of an attack. Paper III (Improving CIDER and Lazarus recovery mechanisms) extends device recovery architectures to handle runtime attacks and improve device availability under network attack conditions, minimizing downtime and improving the way IoT performs recovery during failure.

Papers IV and V address security challenges in 5G networks, particularly in the NWDAF, a key component in the 5G architecture responsible for collecting and analyzing data from various NFs and services, which plays a crucial role in maintaining the availability and performance of 5G services. Paper IV shows that adversaries with device cloning capabilities can significantly degrade NWDAF mobility prediction accuracy from 75% to 40% using just 100 adversarial UEs (out of 10,000 legitimate UEs) in a "tuple jump attack" that creates physically impossible movement patterns. Paper V explores strategies for maintaining NWDAF model performance when the operator uses AutoML to select the model. Under attack conditions, retraining a model with AutoML proves to be less effective than preserving a well-trained initial model that was chosen through a comprehensive AutoML search performed during the initial training phase.

Papers VI and VII focus on vulnerability detection using code language models (code-LMs), addressing the issue of identifying software vulnerabilities that can be exploited to compromise system availability. Paper VI examines the challenges of code-LM-based vulnerability detection for C/C++ and proposes a cumulative model that outperforms binary classification when identifying the most common vulnerabilities. Paper VII investigates the effectiveness of LMs for vulnerability detection in various programming languages, revealing significant performance variations and a weak correlation between code complexity and detection performance.

Collectively, these papers contribute novel solutions, insights, and empirical findings that enhance system availability in the context of IoT, 5G networks, and software systems.

References

[3GP22a] 3GPP. *5G System; Network Data Analytics Services; Stage 3*. Technical Specification (TS) 29.520. Version 17.8.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[3GP22c] 3GPP. *Architecture enhancements for 5G System (5GS) to support network data analytics services*. Technical Specification (TS) 23.288. Version 17.6.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[3GP24a] 3GPP. *5G Security Assurance Specification (SCAS) for the Session Management Function (SMF) network product class*. Technical Specification (TS) 33.515. Version 18.1.0. 3rd Generation Partnership Project (3GPP), Jan. 2024.

[3GP24b] 3GPP. *5G Security Assurance Specification (SCAS); Access and Mobility management Function (AMF)*. Technical Specification (TS) 33.512. Version 18.2.0. 3rd Generation Partnership Project (3GPP), July 2024.

[3GP24c] 3GPP. *Policy and charging control framework for the 5G System (5GS); Stage 2*. Technical Specification (TS) 23.503. Version 19.2.0. 3rd Generation Partnership Project (3GPP), Dec. 2024.

[AC20] M. Ammar and B. Crispo. "Verify&Revive: Secure Detection and Recovery of Compromised Low-end Embedded Devices." In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. Austin, USA: Association for Computing Machinery, 2020, pp. 717–732.

[ACS17] M. N. Aman, K. C. Chua, and B. Sikdar. "A Light-Weight Mutual Authentication Protocol for IoT Systems." In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. 2017, pp. 1–6.

[Alo+19] U. Alon et al. "code2vec: learning distributed representations of code." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019).

[Ant+17a] M. Antonakakis et al. “Understanding the Mirai Botnet.” In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110.

[ARW04] T. Anderson, T. Roscoe, and D. Wetherall. “Preventing Internet denial-of-service with capabilities.” In: *SIGCOMM Comput. Commun. Rev.* 34.1 (Jan. 2004), pp. 39–44.

[Ate+22] H. C. Ates et al. “End-to-end design of wearable sensors.” In: *Nature Reviews Materials* 7.11 (2022), pp. 887–907.

[Avi+04] A. Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing.” In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[Bal99a] T. Ball. “The concept of dynamic analysis.” In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pp. 216–234.

[Bal99b] T. Ball. “The concept of dynamic analysis.” In: *SIGSOFT Softw. Eng. Notes* 24.6 (Oct. 1999), pp. 216–234.

[BB13] A. Bacchelli and C. Bird. “Expectations, outcomes, and challenges of modern code review.” In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 712–721.

[BB22] L. Braz and A. Bacchelli. “Software security during modern code review: the developer’s perspective.” In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 810–821.

[BNM21a] G. Bhandari, A. Naseer, and L. Moonen. “CVEfixes: automated collection of vulnerabilities and their fixes from open-source software.” In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 30–39.

[Bor+23] J. B. Borges et al. “IoT Botnet Detection Based on Anomalies of Multiscale Time Series Dynamics.” In: *IEEE Transactions on Knowledge and Data Engineering* 35.12 (2023), pp. 12282–12294.

[Cas+03] B. Caswell et al. *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

[Cha+22b] S. Chakraborty et al. “Deep Learning Based Vulnerability Detection: Are We There Yet?” In: *IEEE Transactions on Software Engineering* 48.9 (2022), pp. 3280–3296.

[Che+21a] M. Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[Che+21b] X. Chen et al. “Massive Access for 5G and Beyond.” In: *IEEE Journal on Selected Areas in Communications* 39.3 (2021), pp. 615–637.

[Che+23a] Y. Chen et al. “DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection.” In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 654–668.

[CM04] B. Chess and G. McGraw. “Static analysis for security.” In: *IEEE Security & Privacy* 2.6 (2004), pp. 76–79.

[CS24] Q. Chen and N. Sheng. “A novel health monitoring system for vital signs using IoT.” In: *Scientific Reports* 14.1 (2024), p. 19189.

[DAF18] R. Doshi, N. Apthorpe, and N. Feamster. “Machine Learning DDoS Detection for Consumer Internet of Things Devices.” In: *2018 IEEE Security and Privacy Workshops (SPW)*. May 2018, pp. 29–35.

[Dai+24] D. Dai et al. “DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models.” In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1280–1297.

[De +22] I. De Oliveira Nunes et al. “CASU: Compromise Avoidance via Secure Update for Low-End Embedded Systems.” In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’22. San Diego, California: Association for Computing Machinery, 2022.

[Dev+19a] J. Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186.

[Din+25] Y. Ding et al. "Vulnerability Detection with Code Language Models: How Far Are We?" In: *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 469–481.

[El-+09] A. El-Atawy et al. "Adaptive Early Packet Filtering for Defending Firewalls Against DoS Attacks." In: *IEEE INFOCOM 2009*. 2009, pp. 2437–2445.

[EP23] L. F. Eliyan and R. D. Pietro. "DeMi: A Solution to Detect and Mitigate DoS Attacks in SDN." In: *IEEE Access* 11 (2023), pp. 82477–82495.

[FAB12] J. Francois, I. Aib, and R. Boutaba. "FireCol: A Collaborative Protection Network for the Detection of Flooding DDoS Attacks." In: *ACM* (2012).

[Fel+20] A. Feldmann et al. "The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic." In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1–18.

[Fen+20a] Z. Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by T. Cohn, Y. He, and Y. Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[Fio+20] A. Fioraldi et al. "AFL++ : Combining Incremental Steps of Fuzzing Research." In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[Fis+24] G. K. J. Fischer et al. "Evaluation of a Smart Mobile Robotic System for Industrial Plant Inspection and Supervision." In: *IEEE Sensors Journal* 24.12 (2024), pp. 19684–19697.

[FOB05] J. C. Foster, V. Osipov, and N. Bhalla. *Buffer Overflow Attacks*. Syngress Publishing, 2005.

[Gao+23] K. Gao et al. "Your Locations May Be Lies: Selective-PRS-Spoofing Attacks and Defence on 5G NR Positioning Systems." In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 2023, pp. 1–10.

[GHB08] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. "Understanding individual human mobility patterns." In: *nature* 453.7196 (2008), pp. 779–782.

[Gro21] T. C. Group. *DICE Attestation Architecture*. Mar. 2021.

[GSG19] M. Goyal, I. Sahoo, and G. Geethakumari. “HTTP Botnet Detection in IOT Devices using Network Traffic Analysis.” In: *2019 International Conference on Recent Advances in Energy-efficient Computing and Communication (ICRAECC)*. 2019, pp. 1–6.

[GTH15] C. Gehrmann, M. Tiloca, and R. Hoglund. “SMACK: Short message authentication check against battery exhaustion in the Internet of Things.” In: *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. June 2015, pp. 274–282.

[Guo+21] D. Guo et al. “GraphCode{BERT}: Pre-training Code Representations with Data Flow.” In: *International Conference on Learning Representations*. 2021.

[HKV19] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[Hub+20] M. Huber et al. “The Lazarus Effect: Healing Compromised Devices in the Internet of Small Things.” In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’20. Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 6–19.

[IST19] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik. “HEALED: HEaling & Attestation for Low-End Embedded Devices.” In: *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2019, pp. 627–645.

[ISZ17] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni. “SeED: <u>se</u>cure non-interactive attestation for <u>e</u>mbedded <u>d</u>evices.” In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec ’17. Boston, Massachusetts: Association for Computing Machinery, 2017, pp. 64–74.

[Jeo+21a] J. Jeong et al. “Mobility Prediction for 5G Core Networks.” In: *IEEE Communications Standards Magazine* 5.1 (2021).

[JPF17] L. Jäger, R. Petri, and A. Fuchs. “Rolling DICE: Lightweight Remote Attestation for COTS IoT Hardware.” In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. Calabria, Italy, 2017.

[KA24] V. R. Kebande and A. I. Awad. “Industrial Internet of Things Ecosystems Security and Digital Forensics: Achievements, Open Challenges, and Future Directions.” In: *ACM Comput. Surv.* 56.5 (Jan. 2024).

[Kak+24] G. Kakkavas et al. “5G Perspective Of Connected Autonomous Vehicles: Current Landscape and Challenges Toward 6G.” In: *IEEE Wireless Communications* 31.4 (2024), pp. 299–306.

[KB15] D. P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization.” In: *3rd International Conference on Learning Representations (ICLR)*. 2015.

[Kek+18] S. Kekki et al. “MEC in 5G networks.” In: *ETSI white paper* 28 (2018), pp. 1–28.

[KL20] A. Kumar and T. J. Lim. “Early Detection of Mirai-Like IoT Bots in Large-Scale Networks through Sub-sampled Packet Traffic Analysis.” In: *Advances in Information and Communication*. Ed. by K. Arai and R. Bhatia. Cham: Springer International Publishing, 2020, pp. 847–867.

[Kol+17] C. Kolias et al. “DDoS in the IoT: Mirai and Other Botnets.” In: *Computer* 50.7 (2017), pp. 80–84.

[Kom+21] O. Kompougias et al. “IoT Botnet Detection on Flow Data using Autoencoders.” In: *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. 2021, pp. 506–511.

[KRS09] A. Kadav, M. J. Renzelmann, and M. M. Swift. “Tolerating hardware device failures in software.” In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 59–72.

[Law+24] N. P. Lawrence et al. “Machine learning for industrial sensing and control: A survey and practical perspective.” In: *Control Engineering Practice* 145 (2024), p. 105841.

[LE20] J. Lumbroso and J. Evans. “Making Manual Code Review Scale.” In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE ’20. Portland, OR, USA: Association for Computing Machinery, 2020, p. 1390.

[LeC+89] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition.” In: *Neural Computation* 1.4 (1989), pp. 541–551.

[Li+18a] Z. Li et al. “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection.” In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018.

[Li+24] H. Li et al. “Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach.” In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024).

[Liu+18b] Z. Liu et al. “Practical Proactive DDoS-Attack Mitigation via Endpoint-Driven In-Network Traffic Control.” In: *IEEE/ACM Trans. Netw.* 26.4 (Aug. 2018), pp. 1948–1961.

[Liu+20] Y. Liu et al. *Ro{BERT}a: A Robustly Optimized {BERT} Pretraining Approach*. 2020.

[LKG13] S. B. Lee, M. S. Kang, and V. D. Gligor. “CoDef: Collaborative Defense against Large-Scale Link-Flooding Attacks.” In: CoNEXT ’13. Santa Barbara, California, USA: ACM, 2013, pp. 417–428.

[LL05] V. B. Livshits and M. S. Lam. “Finding security vulnerabilities in java applications with static analysis.” In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM’05. Baltimore, MD: USENIX Association, 2005, p. 18.

[Mag+24] A. Maghsoudnia et al. “Ultra-Reliable Low-Latency in 5G: A Close Reality or a Distant Goal?” In: *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. HotNets ’24. Irvine, CA, USA: Association for Computing Machinery, 2024, pp. 111–120.

[Mea08] D. H. Meadows. “Thinking in systems: A primer.” In: *Sustainability Institute* (2008).

[Mei+18] Y. Meidan et al. “N-BaIoT—Network-Based Detection of IoT Botnet Attacks Using Deep Autoencoders.” In: *IEEE Pervasive Computing* 17.3 (2018), pp. 12–22.

[Mer88] R. C. Merkle. “A Digital Signature Based on a Conventional Encryption Function.” In: *Advances in Cryptology — CRYPTO ’87*. Ed. by C. Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.

[Mil+95] B. P. Miller et al. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1995.

[MK22] S. N. Mishra and M. Khatua. “Achieving Hard Reliability in RPL for Mission-Critical IoT Applications.” In: *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*. 2022, pp. 1–6.

[MP21] N. Mishra and S. Pandya. “Internet of Things Applications, Security Challenges, Attacks, Intrusion Detection, and Future Visions: A Systematic Review.” In: *IEEE Access* 9 (2021), pp. 59353–59377.

[MR05] J. Mirkovic and P. Reiher. "D-WARD: a source-end defense against flooding denial-of-service attacks." In: *IEEE Transactions on Dependable and Secure Computing* 2.3 (July 2005), pp. 216–232.

[MW23] S. Mallissery and Y.-S. Wu. "Demystify the Fuzzing Methods: A Comprehensive Survey." In: *ACM Comput. Surv.* 56.3 (Oct. 2023).

[Nam+24] D. Nam et al. "Using an LLM to Help With Code Understanding." In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024.

[NB18] S. Nõmm and H. Bahşi. "Unsupervised Anomaly Based Botnet Detection in IoT Networks." In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, pp. 1048–1053.

[Nee93] R. M. Needham. "Denial of service." In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. CCS '93. Fairfax, Virginia, USA: Association for Computing Machinery, 1993, pp. 151–153.

[Ngu+20] M.-D. Nguyen et al. "Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities." In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 47–62.

[Noa+22] M. Noaman et al. "Challenges in Integration of Heterogeneous Internet of Things." In: *Sci. Program.* 2022 (Jan. 2022).

[Nun+20] I. D. O. Nunes et al. "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 771–788.

[OV23] A. Oprea and A. Vassilev. *Adversarial machine learning: A taxonomy and terminology of attacks and mitigations (draft)*. Tech. rep. National Institute of Standards and Technology, 2023.

[OZ23] L. Osler and D. Zahavi. "Sociality and embodiment: Online communication during and after Covid-19." In: *Foundations of Science* 28.4 (2023), pp. 1125–1142.

[PCH16] Z. Pi, J. Choi, and R. Heath. "Millimeter-wave gigabit broadband evolution toward 5G: fixed access and backhaul." In: *IEEE Communications Magazine* 54.4 (2016), pp. 138–144.

[PL01] K. Park and H. Lee. "On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets." In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 15–26.

[PME19] R. Paudel, T. Muncy, and W. Eberle. “Detecting DoS Attack in Smart Home IoT Devices Using a Graph-Based Approach.” In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 5249–5258.

[Rad+19] A. Radford et al. “Language models are unsupervised multitask learners.” In: *OpenAI blog* 1.8 (2019), p. 9.

[RFB17] B. Rashidi, C. Fung, and E. Bertino. “A Collaborative DDoS Defence Framework Using Network Function Virtualization.” In: *IEEE Transactions on Information Forensics and Security* 12.10 (2017), pp. 2483–2497.

[Rob57] C. C. Robusto. “The cosine-haversine formula.” In: *The American Mathematical Monthly* 64.1 (1957), pp. 38–40.

[Ron+17] E. Ronen et al. “IoT Goes Nuclear: Creating a ZigBee Chain Reaction.” In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 195–212.

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask).” In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 317–331.

[Sad+18a] C. Sadowski et al. “Lessons from building static analysis tools at Google.” In: *Commun. ACM* 61.4 (Mar. 2018), pp. 58–66.

[Sad+18b] C. Sadowski et al. “Modern code review: a case study at google.” In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 181–190.

[Sal+19] Y. Saleem et al. “Internet of Things-Aided Smart Grid: Technologies, Architectures, Applications, Prototypes, and Future Research Directions.” In: *IEEE Access* 7 (2019), pp. 62962–63003.

[SB14] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[Sca+09] F. Scarselli et al. “The Graph Neural Network Model.” In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80.

[SES21] Y. E. Sagduyu, T. Erpek, and Y. Shi. “Adversarial machine learning for 5G communications security.” In: *Game Theory and Machine Learning for Cyber Security* (2021), pp. 270–288.

[Sfo+16] A. Sforzin et al. “RPiDS: Raspberry Pi IDS — A Fruitful Intrusion Detection System for IoT.” In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. 2016, pp. 440–448.

[Sha+15b] A. Shameli-Sendi et al. “Taxonomy of Distributed Denial of Service mitigation approaches for cloud computing.” In: *Journal of Network and Computer Applications* 58 (2015), pp. 165–179.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014.

[She+22] M. J. Shehab et al. “5G Networks Towards Smart and Sustainable Cities: A Review of Recent Developments, Applications and Future Perspectives.” In: *IEEE Access* 10 (2022), pp. 2987–3006.

[Shi+24] K. Shi et al. *Natural Language Outlines for Code: Literate Programming in the LLM Era*. 2024. arXiv: 2408.04820 [cs.SE].

[SM17] H. Sinanović and S. Mrdovic. “Analysis of Mirai malicious software.” In: *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2017, pp. 1–5.

[Son+10] C. Song et al. “Limits of Predictability in Human Mobility.” In: *Science* 327.5968 (2010), pp. 1018–1021. eprint: https://www.science.org/doi/pdf/10.1126/science.1177170.

[Suz+20] K. Suzaki et al. “Reboot-Oriented IoT: Life Cycle Management in Trusted Execution Environment for Disposable IoT devices.” In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC ’20. Austin, USA: Association for Computing Machinery, 2020, pp. 428–441.

[Svy+20] A. Svyatkovskiy et al. “Intellicode compose: Code generation using transformer.” In: *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 1433–1443.

[Tag+24] C. Tagliaro et al. “Large-Scale Security Analysis of Real-World Backend Deployments Speaking IoT-Focused Protocols.” In: *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’24. Padua, Italy: Association for Computing Machinery, 2024, pp. 561–578.

[Tao+21] Z. Tao et al. "{DICE*}: A Formally Verified Implementation of {DICE} Measured Boot." In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1091–1107.

[Tao+23] Q. Tao et al. "Clinical applications of smart wearable sensors." In: *iScience* 26.9 (2023), p. 107485.

[Tea24] G. Team. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL].

[Tho21] P. Thomson. "Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity." In: *Queue* 19.4 (Sept. 2021), pp. 29–41.

[Tym17] Y. Tymchuk. "The False False Positives of Static Analysis." In: (2017).

[Vas+17] A. Vaswani et al. "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. New York, NY, USA: Curran Associates, Inc., 2017.

[Wan+21c] Y. Wang et al. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens et al. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708.

[Wel05] M. Welzl. *Network congestion control: managing internet traffic*. John Wiley & Sons, 2005.

[Xia+23] F. Xiao et al. "Over-the-Air Adversarial Attacks on Deep Learning Wi-Fi Fingerprinting." In: *IEEE Internet of Things Journal* 10.11 (2023), pp. 9823–9835.

[Xu+19] M. Xu et al. "Dominance as a New Trusted Computing Primitive for the Internet of Things." In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1415–1430.

[Yan+22] Y. Yang et al. "A Survey on Deep Learning for Software Engineering." In: *ACM Comput. Surv.* 54.10s (Sept. 2022).

[Yua+22] Y. Yuan et al. "Insight of Anomaly Detection with NWDAF in 5G." In: *2022 International Conference on Computer, Information and Telecommunication Systems (CITS)*. 2022, pp. 1–6.

[ZD19] H. Zhang and L. Dai. "Mobility Prediction: A Survey on State-of-the-Art Schemes and Future Applications." In: *IEEE Access* 7 (2019), pp. 802–822.

[Zha+24] Z. Zhang et al. “Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code.” In: *Transactions on Machine Learning Research* (2024).

[Zho+19a] Y. Zhou et al. “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.” In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

[ZL22] T. Zheng and B. Li. “Poisoning Attacks on Deep Learning based Wireless Traffic Prediction.” In: *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 2022, pp. 660–669.

Included Publications

CLI-DOS: Collaborative Counteraction against Denial of Service in the Internet of Things

Abstract

Internet of Things (IoT) is the next generation of network scenario where billions of devices are connected to the internet and capable to communicate with each other. However, they are especially vulnerable to Denial of Service (DoS) attack, which typically launched by overloading the server with bogus messages. Hence, unnecessary computation is performed by the IoT units, which can seriously worsen their performance or even make them unable to serve legitimate requests. Existing mechanisms for the detection and prevention of DoS attacks only either make it more difficult to perform massive DoS attacks or completely block the legitimate requests to be served. In this paper, we present CLI-DOS, a collaborative mechanism between Gateway and IoT units to counteract Denial of Service attacks. CLI-DOS combines an efficient attack-detection principle using short MAC value with a more effective defense strategy. In an under attack situation, we leverage the IoT gateway for preventing the attack messages to make contact with the IoT units while opening up possibilities to set-up connection with legitimate external entities. We evaluate the performance of our proposal from several standpoints. In conclusion, CLI-DOS offers a lower resource utiliza-

Syafiq Al Atiiq, Christian Gehrmann. "CLI-DOS: Collaborative Counteraction against Denial of Service in the Internet of Things". In 18^{th} *International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2020, Austin, TX, USA*. pp. 1-6, IEEE.

tion in an under attack IoT units while preserving service availability to the best possible extent.

1 Introduction

Internet of Things (IoT) is commonly referred to an increasing trend in information technology en route to *networked society*, where all devices are interconnected and able to harness the connection to the internet [Kor+10]. In some cases, these devices are even used for prominent and important tasks such as health monitoring, industrial automation, or energy management. However, many IoT devices are built upon a constrained hardware platform, which has limited resources of CPU, memory, or even power source compared to e.g. personal computers. That being said, keeping the availability of the devices as long as possible becomes a very important task.

IoT units with a direct connection to the internet are prone to be exposed to DoS attack. The common strategy used by the adversaries consist in flooding the server devices with invalid messages such that the server becomes exhausted and unable to serve legitimate requests. As most IoT units are battery-powered, the impact of DoS attack to those devices is considerably severe which can greatly reduce the lifetime of the IoT units or even make them out of battery at once.

A number of countermeasure strategies have been proposed to counteract DoS attacks. As explained by Wang [WJS07], strategies can be classified into two types, namely *router-based* and *host-based*. First, *router-based* approach addresses the problem by deploying a built-in defense mechanism running on the intermediate nodes between client and server. On Cisco routers [Cis08], this include for instance: reverse routing path verification and IP address filtering. As for the *host-based* solution, IoT units are designed to use cryptographic protocols e.g. DTLS [RM12] and IKE [Kau05]. As these protocols use a stateless *cookie* exchange, launching DoS attack with spoofed IP addresses towards IoT units would be difficult. On the other hand, the cryptographic operations employed in the IoT units can be quite expensive in terms of power consumption at the set-up phase. Furthermore, as mentioned in the standard [RM12], a *cookie* does not provide any defense mechanism against DoS attack launched from valid IP addresses.

Another approach [Hum+13] is by forcing the adversaries to solve a computational demanding puzzle as a requirement to proceed to the handshake phase. However, with an increasing power of modern GPU, the puzzles from the victim can be easily solved [NM16]. Furthermore, a more advanced strategy to determine the validity of incoming request is by including a short symmetric key based MAC value in the packet header when both parties, client and server, are trying to establish a connection [TGS17][GTH15][GS13]. While this is very efficient to detect the occurrence of DoS attack in the IoT units, such strategy will at the same time completely prevents the legitimate clients to reach the IoT units. An adaptive solution, called SARDOS [THA18] has been proposed to counteract DoS attack

while preserving service availability from legitimate clients. Even though SARDOS mechanism is already good enough to prevent the battery drain by adaptively changing the server state, but the computational burden to check the validity of the incoming requests is still inside the IoT units. Aside from shutting down the interface, there is no mechanism in SARDOS to drop the attack messages even before reaching the IoT units.

To fill the gap, in this paper we introduce CLI-DOS, a collaborative mechanism between the IoT units and the gateway to counteract DoS attacks. In particular, we leverage an efficient DoS detection mechanism from the previous work [TGS17][GTH15][GS13] with a novel collaboration protocol between the IoT units and the gateway at the border of IoT wireless domain. Our rationale is, it would be better to offload some computational expensive process of filtering attack messages from the IoT units to a much more powerful gateway that sits close to the IoT wireless domain. The gateway will have the capability to filter the message ID of the CoAP request based on the acceptable range provided by the victim. This way, we are able to protect the wireless IoT units from battery drain attack, even when the adversaries changing their IP address many times, *while at the same time* allowing legitimate request to be served. CLI-DOS will to some extent lessen the capabilities of IoT units to communicate with external entities, however, it will not completely block the communication from serving the critical messages even when under heavy attacks.

We have made a proof-of-concept implementation of CLI-DOS for the CoAP [SHB14] protocol using ContikiOS [DGV04] inside resource-constrained CC2538 platform. Yet, CLI-DOS is designed with the general mindset and can be implemented at any communication protocol stack. On the gateway side, we have made an implementation using $u32$ module of netfilter [Net19] inside the kernel in a way that it can parse the CoAP message ID as one of our requirement. Also, the design of our protocol in the gateway side is general so that it can be implemented on any linux flavor, either in *user-space* or *kernel-space*. As some computational burden of filtering attack messages has been off-loaded to the gateway unit, our results show that CLI-DOS has a lower resource utilization in terms of energy consumption at the IoT units compared to the vanilla CoAP implementation and pure IP blocking mechanism at the gateway. In addition, CLI-DOS has efficiently dropped the bogus messages even before reaching the under-attack IoT units while still serving legitimate requests from external entities at the same time.

The rest of the paper is organized as follows. We discuss related works in Section II and background concepts in Section III. We provide the application scenario in Section IV. Section V presents collaborative solution against DoS, while in Section VI we provide a performance evaluation of CLI-DOS. Section VII draws our conclusions and anticipate future works.

2 Related Work

Denial of Service is a security menace, as defined at [TGS17], in which the aim is to (partially or completely) disrupt the service of server(s). The victim is usually forced to use its resource in a high fashioned utilization. This include for instance CPU, memory, and network bandwidth as a means to make it less responsive or even unable to serve legitimate request. The use of DoS to attack can be classified into *Distributed Denial of Service* [Mir+04] if the attacker sources are using many machines. This usually done by creating a well coordinating master and slave nodes, in which the master act as a leader.

To counteract DoS, there have been several ideas proposed. As classified by Wang [WJS07], there are two types of DoS counteraction, namely *router-based* and *client-based*. A router-based mechanism solve the problem by identifying address passed by the attacker [Li+04], thanks to its built-in mechanism planted on every router on the path from the client to server. This particularly efficient for a network that is managed by single entity where we have the access to all the router. On the other hand, router-based solution normally need the involving router to implement *Probabilistic Packet Marking (PPM)* and coordinating mechanism between them. This would lead to the inconvenience of the person in charge as they need to maintain many coordinating nodes with additional complexity. Seidel proposes a protocol validation to complicate remote DoS attack [SKM19]. While this is a good starting point to offload the filtering mechanism to the border router, we believe that a further step to determine the validity of request message should be taken into account.

Host-based approach counteract DoS by building a mechanism at the end nodes, which does not need a coordination function between entities. Several examples of *host-based* approach can rely on resource management schemes [QPP02][BF99]. Another proposed solution that can be classified into *host-based* leverages puzzle [XR03]. In order to legitimately access the server, a client has to solve puzzles which has several level of difficulty. While this can prevent a DoS to some extent, but with an increasing power of modern GPU, a puzzle from a resource constrained device can easily be solved [NM16].

Our solution in this paper falls between *router-based* and *host-based*. We leverage the simplicity of setting up *host-based* approach while still maintain contact with the router to get better quality and resource to filter bogus messages. To get a better segregation on valid and invalid messages on the server side, we leverage SMACK [GTH15], a host-based countermeasure to DoS that make use of short Message Authentication Code (MAC). The server analyze the embedded short MAC on the request message sent to them and instantly get a classification of the validity of the message.

The additional value of our proposed solution compared to traditional DoS mitigation relies on the coordination function between victim server and the router on the same network. As the router is usually more powerful in terms of resource,

we move some computational burden of filtering messages to them, not only using IP based blocking but also a mechanism based on message ID of CoAP message. This way, even the adversaries changing its IP address, we can preserve the battery on the server side while maintaining the service to the legitimate user to the best possible extent.

3 Background

3.1 CoAP

The Constrained Application Protocol [SHB14] is a lightweight communication protocol, specifically designed for resource constrained nodes and networks. To some extent, it is similar to HTTP but for affordable and lightweight type of communication. CoAP's main purpose is to be used for Internet of Things environment, where machine to machine communication take place. Home automation, factory monitoring, and connected vehicles are some examples of where CoAP is suited.

Unlike HTTP, CoAP uses UDP as its transport protocol, meaning that it has less overhead in terms of the amount of messages exchanged between parties. With regards to security, CoAP does not provide any mechanism, but rather offload it to other components. The most recommended one is DTLS, where it can provide secure communication, e.g. authentication, integrity, and confidentiality of the messages that's exchanged between nodes.

3.2 SMACK

SMACK [GTH15], correspond to Short Message Authentication CheckK, is a security mechanism to identify a validity of incoming messages in the server. The way SMACK works is by probing into a short and lightweight Message Authentication Code embedded in the message. The output of SMACK processing is a verdict whether a message is a valid request (and proceed for further processing) or not (discarded).

SMACK is designed to work well with CoAP, in which it uses the Token field to embed short MAC in the message. By message structure, there is not much modification of the CoAP header, except the Token field itself. In advance of sending the message, client generates 16-bits random *Request ID R* as a means to match request and respective response messages. Following *R*, client computes 16-bits *short MAC SM*. Then, the client put *R* and *SM* in the Request ID and Validity check field respectively, as depicted in figure 1.

Upon receiving the message, the server computes the short MAC of the incoming message and compare the value with the provided short MAC of the request message.

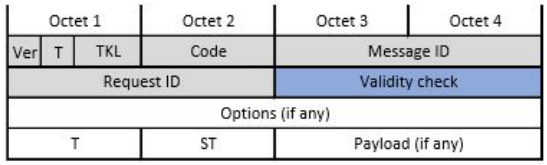

Figure 1: CoAP Message with Short MAC

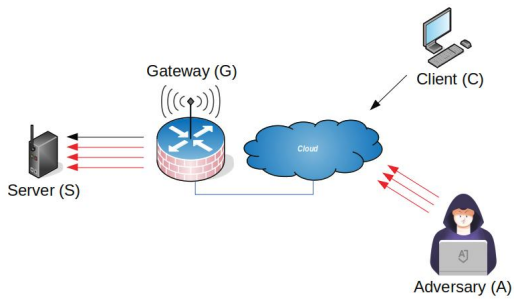

Figure 2: Application Scenario

4 Application Scenario

Henceforward, we examine the scenario in Figure 2. It consists of a Server S and a Client C exchanging message over CoAP protocol. At the same time, an adversary A launch a DoS attack by sending large number of invalid CoAP messages towards S. The aim of the attack messages is to make S worthlessly commit resources in a way that is not serving actual legitimate request. A also has the capability to spoof its IP address as many times as he wants. This way, S could be endangered by becoming less responsive or even running out of energy due to battery drain.

A Gateway G sits between C and S. The gateway carry out the job as a message forwarder between C and S. We assume that the connection between G and S are in the wireless communication channel, meanwhile communication towards C is a wired connection. The adversary A always come from the wired connection domain. The case where A is coming from the wireless domain is out of scope of this paper.

At any times, S should be able to identify legitimacy of the request message. A legitimate request from C is always processed and replied with a CoAP response. On the other hand, an illegitimate request from A should be detected and S should further act (i.e. report to Gateway G) as explained in chapter 5. In this paper, the capability to identify legitimacy of the message come from the implementation of SMACK [GTH15].

We assume G is secure enough, such that it is unattainable to be compromised. It is possible to design G as a distributed gateway to get a better reliability as

described in [And10][Bir12]. The work to design such system is out of scope of this paper. Nevertheless, we assume that G is robust and reliable such that no matter how high the DoS magnitude, does not kill G.

5 Collaborative Counteraction Against DoS

In this section, we present CLI-DOS, the Collaborative Counteraction against DoS. The fundamental motivation of CLI-DOS lies in two following arguments: (i) It is possible to offload computational expensive filtering at the server to a much more powerful gateway, while at the same time (ii) Allowing legitimate request to be served in a best effort manner. CLI-DOS leverages the Message ID field in the CoAP header to determine the validity of subsequent CoAP requests. This will allow the gateway to detect IP spoofing events from the adversaries. CLI-DOS will to some extent reduce the communication capabilities of the server, however it will not completely stop the server to perform critical communication task even under heavy DoS attacks. The following subchapter explain the detailed procedures of CLI-DOS both from the server and gateway side.

5.1 Procedures from The Server Perspective

At any times, the server measures the number of invalid message(s) m for every time period t. In a normal operation, if there is no DoS attack occured, S resets m to 0 when t is over. However, if the number of invalid messages m exceeds a predefined threshold h, S contacts G and sends a request r to start DoS protection. Let ID_{qa} be the message ID accepted by SMACK as a valid request [GTH15], and $\overline{ID_q}$ as a set of accepted message ID of size x, then Algorithm 1 summarize S procedures for every time period t.

When a newer valid request arrived, $\overline{ID_q}$ should be updated accordingly. For example, let x be 20, once ID_{q1} arrived, $\overline{ID_q}$ would contains $\{ID_{q2}, ID_{q3}, ID_{q4}, ..., ID_{q21}\}$. Since CoAP runs on top of UDP, it is possible that request messages arrived out of order. For example, instead of ID_{q1}, it is ID_{q13} whose arrived first. Then $\overline{ID_q}$ would delete ID_{q13} and $\{ID_{q1}, ..., ID_{q(13-(20/2))}\}$, and append $1 + (13 - 20/2)$ additional message ID's at the end of the list. Hence, $\overline{ID_q}$ would become $\{ID_{q4}, ID_{q5}, ..., ID_{q12}, ID_{q14}, ..., ID_{q23}, ID_{q24}\}$. However, if the message ID is less than $x/2$, $\overline{ID_q}$ will just delete ID_{qa}, and append 1 new message ID at the end of the list.

Request r includes the following information:

- The latest state of set $\overline{ID_q}$.
- A list of recorded source IP addresses in which the Server S wants to block.
- Maximum number of allowed IP packets per second, p.

Algorithm 1 Procedure at S

```
1: ID_q = {ID_q1, ID_q2, ..., ID_qx}
2: m = 0
3: while time < t do
4:     <Receive and check short MAC>
5:     if CoAP Request Valid and ID_qa ∈ ID_q then
6:         ID_q.delete(ID_qa)
7:         if a < (x/2) then
8:             ID_q.append(ID_q(x+1))
9:         else
10:            ID_q.delete(ID_q1, ..., ID_q(a−(x/2)))
11:            ID_q.append(ID_q(x+1), ..., ID_q(x+1+(a−(x/2))))
12:        end if
13:    else if CoAP Request Invalid then
14:        m = m + 1
15:        if r has not been sent and m > h then
16:            <Send r to G, starts s_1 and turn off radio>
17:        else
18:            <Send m_f to G >
19:        end if
20:    end if
21: end while
22: m = 0
```

- Threshold value h, denotes the maximum number of invalid messages for a time period t.
- A sleep period, s.

Once S goes into the sleep mode, it initializes an internal clock s_1, which then followed by turning off its radio communication. On the other hand, if r has been sent to G, S only sends m_f, a flag message informing G of each individual invalid request arrived at S.

5.2 Procedures from The Gateway Perspective

When there is no request r from S, G operates normally by forwarding all request messages from C to S. However, when r received, G initialize s_2, a timer to indicate when S is going to wake up again. This step marks G to operates on a more strict rules on who can send CoAP request towards S.

During this period, all packets which have the destination address of S are checked with respect to their higher level protocol, such that: i) all packets with the messages ID not in the list of $\overline{ID_q}$, ii) all packets with forbidden IP addresses, and iii) all packets but CoAP, are dropped.

G measures the number of dropped and accepted packets per second targetting S, denoted as d and a respectively. When period s is over, S turned on its radio

Algorithm 2 Procedure at G during attack period

```
1: <Receive r from S >
2: ID_q = {ID_q1, ID_q2, ..., ID_qx}
3: <Starting timer s_2 >
4: <Apply filtering mechanism as requested by S >
5: a = 0 and d = 0
6: <Initiate messages buffer with the length x >
7: while s_2 is not expired do
8:     <Receive CoAP request packet from C >
9:     if ID_qa ∈ ID_q then
10:        < a = a + 1 and put the packet into buffer>
11:    else
12:        < d = d + 1 and discard the packet>
13:    end if
14: end while
15: if a < p then
16:    <Forwards all queued packet to S >
17: else
18:    <More severe attack happens>
19:    <Sends warning w to the system responsible>
20: end if
```

communication again, and notify G. When G get notified by S, it checks whether $s_2 > s$ as well as if $a < p$, where p is a threshold determines the maximum allowed load on S. If both conditions are met, G starts forwarding sequentially all queued packets targetting S until the packet buffer is empty. G then sets a to 0 and reinitialized s_2 to 0. Otherwise, when $s_2 > s$, G would assumes that a more severe attack happens and sends a warning message w to the system responsible. Algorithm 2 summarizes G behavior during the attack period.

Also, when filtering mechanism kicked in, G should have a MID list processing, regardless of the state of S. When a request with MID ID_{qa} received, G checks if it is in the $\overline{ID_q}$ or not. If yes, then G puts ID_{qa} into a temporary list $\overline{ID_t}$ where the size is adjustable. Meanwhile, G initialize a timer t_w, where it expects to receive an ominous message from S, saying that CoAP request with ID_{qa} is an attack. If it does not happen until t_w expired, G assumes that the request is valid, and delete ID_{qa} from $\overline{ID_t}$, followed by the adaptation of $\overline{ID_q}$ similar to the process explained in section 5.1. Algorithm 3 summarizes the G message ID processing, which running parallel with Algorithm 2 as a non-blocking operations.

6 Experimental Evaluation

In this section, we discuss the setup, scenario, and results of the evaluation of CLI-DOS. We compare CLI-DOS with Vanilla CoAP and pure IP based protection at G. We have developed a prototype of CLI-DOS on Contiki Operating System [DGV04]. The new developed firmware is tested on Zolertia Firefly platform

Algorithm 3 Procedure at G at any times

```
1: ID_t = {}
2: <Receive CoAP with MID ID_qa from C >
3: if ID_qa ∈ ID_q then
4:     ID_t.append(ID_qa)
5:     <Starts t_w >
6:     if t_w expired then
7:         <CoAP message with ID_qa is valid>
8:         ID_t.delete(ID_qa) and ID_q.delete(ID_qa)
9:         if a < (x/2) then
10:            ID_q.append(ID_q(x+1))
11:        else
12:            ID_q.delete(ID_q1, ..., ID_q(a−(x/2)))
13:            ID_q.append(ID_q(x+1), ..., ID_q(x+1+(a−(x/2))))
14:        end if
15:    else if m_f received before t_w expired then
16:        <CoAP message with ID_qa is an attack>
17:        ID_t.delete(ID_qa)
18:    end if
19: else
20:    <Drop CoAP Request>
21: end if
```

[Zol19], which has the following specifications : CC2538 radio chipset, 32 kB RAM, 512 kB of flash ROM.

Our experiment follows the scenario depicted in Fig 2, where S is connected to G using IPv6 over IEEE 802.15.4 [CMW17] transmission running 6LoWPAN stack. S is the Zolertia Firefly [Zol19] device running extended CoAP server equipped with CLI-DOS. G is a dedicated VM running contiki rpl-border-router and connected to a slip radio through serial connection. The combination of rpl-border-router and slip-radio makes it possible to bridge the wired domain with the 6LoWPAN protocol in the wireless domain. G, C, and A runs on a native linux and performs the computation on an identic but fully isolated VM with native IPv6 connectivity. All of the VM's equipped with 2 GB Memory and 1 core vCPU.

We perform three experiments, namely 1) E_PLAIN, where S running Vanilla CoAP implementation, and G acts as a forwarder, where all the packets are forwarded to S. 2) E_SMACK_IP, where S has SMACK [GTH15] and able to shuts down its interface and report to G the source IP of A, when certain limit of attack is reached. For simplicity, we made the limit same as h value at E_CLI-DOS_MID. 3) E_CLIDOS_MID, where S running CLI-DOS with message ID filtering mechanism. To this end, aside from source IP filtering, G has the capability to inspect CoAP message based on its MID, as we explained in the previous section. One experiment performed by running 100 Request-Response exchange messages between C and S. One experiment considered as done only if the last

request message has been replied by the S, and received by C. It is possible to lost the message if S is fully occupied by A. In this case, the experiment considered to be done when the last retransmission message on the last request has been transmitted. C sends similar request message every 2 sec, with the size of 28 bytes.

We examine CLI-DOS performance with five different attack settings, that is: 0, 5, 10, 15, and 20 msg/s. We run each experiments 5 times for each attack settings, and then calculate the mean value. A sends 19-bytes CoAP request as the attack messages in all experiments. Also, during our evaluation, we set the threshold h to be 9 msg/s and sleep time s to be 60 sec. These values are determined heuristically through repeated test and evaluations. The performances are measured in terms of i.) energy consumption at S, and ii.) RTT from C.

6.1 Results

Fig 3 shows average energy consumption at S for 5 different attack settings. When there is no attack, each of the experiments shows little difference on the energy usage. It means that SMACK as a message validator, does not have considerable amount of additional overhead.

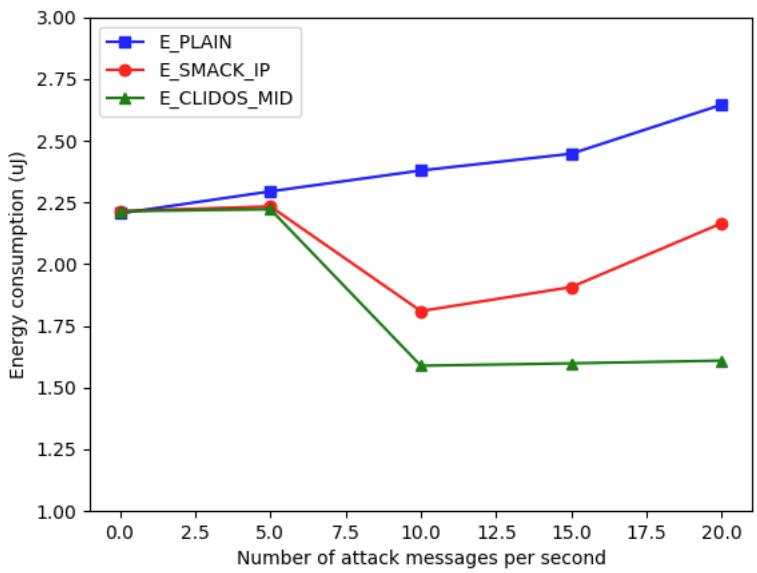

Figure 3: Energy Consumption of each attack configuration

However, when the attack continue to rise to 5 msg/s, E_PLAIN uses 2.29 μJ, E_SMACK_IP uses 2.23 μJ, while E_CLIDOS_MID uses 2.22 μJ. Intuitively, E_SMACK_IP and E_CLIDOS_MID should show lower energy consumption than shown in the graph. However, this is happened because S considers 5 msg/s as a non-malignant attack. The reason is because we set the threshold h to be 9

attack messages per second (we would get a similar result with a different threshold selection). Hence, S would not send any request r towards G to start filtering protection. At 20 msg/s, E_PLAIN reaches the highest point 2.65 μJ, while E_SMACK_IP uses 2.17 μJ. On the other hand, E_CLIDOS_MID comfortably uses only 1.61 μJ energy during the experiment. It means that even though E_SMACK_IP performs IP filtering at G, it is not enough to protect S once A spoof its address. Meanwhile, as E_CLIDOS_MID performs deeper payload inspection (based on MID) for each incoming request, it has better performance because less bogus messages are arrived at S.

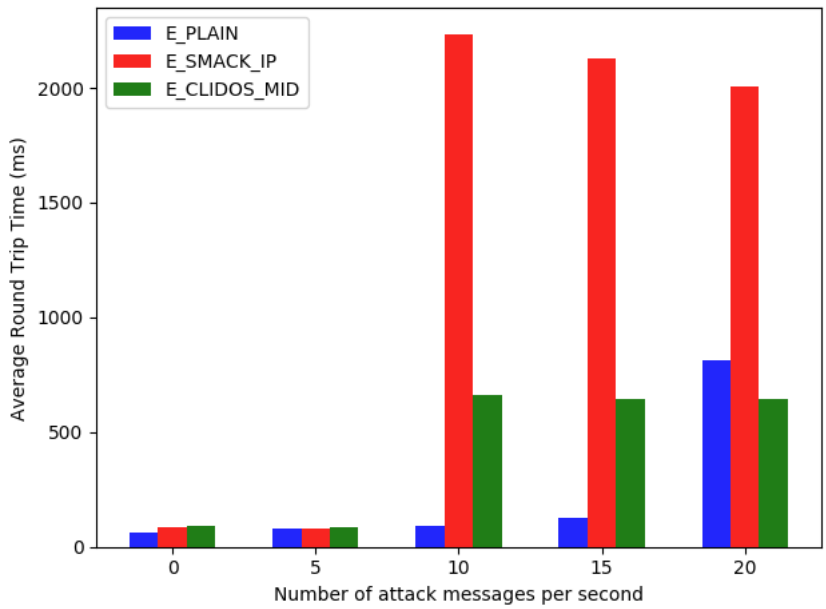

Figure 4: Average Round Trip Time

Fig 4 shows the average RTT for each different attack settings in all experiments. We can see that during 0 and 5 msg/s (benign attack), there is not much different between them. However, when the attack goes up to 20 msg/s, E_CLIDOS_MID (645.28 ms) performs better than E_SMACK_IP (2005.95 ms) and E_PLAIN (811.28 ms). E_SMACK_IP took longer time because it shuts down its interface much longer than E_CLIDOS_MID due to IP address spoofing at the advesary, which then followed by subsequent attack. On the other hand, E_CLIDOS_MID could serve the request better than others, due to MID blocking at G such that adversary A could not reach S even by spoofing their IP address. As Fig 4 is average value, E_SMACK_IP and E_CLIDOS_MID's are get affected by sleep period s of the victim. But, it is only a fraction of the messages need to wait longer to get reply due to s.

7 Conclusion and Future Work

This paper has presented CLI-DOS, a Collaborative Counteraction against Denial of Service. Through this collaboration, it is possible to off-load the computational expensive packet filtering from the IoT units to a much more powerful gateway. This way, CLI-DOS has effectively prevent the IoT units from worthlessly commit of resources during DoS attack, while at the same time serving legitimate requests to the best possible extent. The solution works without requiring any trust between the IoT unit and gateway except for sharing filtering parameters, i.e. SMACK shared secrets are *not* shared. We have experimentally evaluated CLI-DOS through our prototype on Contiki OS running on Zolertia Firefly platform. The results show that, when under heavy attack, CLI-DOS successfully preserve the energy consumption, such that IoT units does not have to deal with the attack messages and can serve the legitimate client better. This come at the price of some longer RTT for some request packet, due to waiting time of the server when they shut down their interface. Future works will include evaluation of larger IoT networks and under different wireless channel conditions. Also, scenarios with more powerful adversaries is left to consider.

Acknowledgements

Work supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research as well as the EU H2020 project CloudiFacturing under grant 768892.

References

[And10] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing Inc., 2010.

[BF99] N. Bhatti and R. Friedrich. "Web server support for tiered services." In: *IEEE Network* 13.5 (Sept. 1999), pp. 64–71.

[Bir12] K. P. Birman. *Guide to Reliable Distributed Systems. Building High-Assurance Applications and Cloud-Hosted Services*. springer, 2012.

[Cis08] Cisco. *Cisco Guide to Defending Against Distributed DoS Attacks*. 2008.

[CMW17] S. Chakrabarti, G. Montenegro, and J. Woodyatt. *IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) ESC Dispatch Code Points and Guidelines*. RFC 8066. Feb. 2017.

[DGV04] A. Dunkels, B. Gronvall, and T. Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors." In: *29th Annual IEEE International Conference on Local Computer Networks*. Nov. 2004.

[GS13] C. Gehrmann and G. Selander. *Methods, Nodes and Computer Programs for Reduction of Undesired Energy Consumption of a Server Node*. 2013.

[GTH15] C. Gehrmann, M. Tiloca, and R. Hoglund. "SMACK: Short message authentication check against battery exhaustion in the Internet of Things." In: *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. June 2015, pp. 274–282.

[Hum+13] R. Hummen et al. "Tailoring end-to-end IP security protocols to the Internet of Things." In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. Oct. 2013, pp. 1–10.

[Kau05] C. Kaufman. *Internet Key Exchange (IKEv2) Protocol*. RFC 4306. Dec. 2005.

[Kor+10] G. Kortuem et al. "Smart objects as building blocks for the Internet of things." In: *IEEE Internet Computing* 14.1 (Jan. 2010), pp. 44–51.

[Li+04] J. Li et al. "Large-scale IP traceback in high-speed Internet: practical techniques and theoretical foundation." In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. May 2004, pp. 115–129.

[Mir+04] J. Mirkovic et al. *Internet Denial of Service: Attack and Defense Mechanisms (Radia Perlman Computer Networking and Security)*. 2004.

[Net19] Netfilter. *Netfilter*. 2019. URL: `https://www.netfilter.org/`.

[NM16] Nithun Chand O and S. Mathivanan. "A survey on resource inflated Denial of Service attack defense mechanisms." In: *2016 Online International Conference on Green Engineering and Technologies (IC-GET)*. Nov. 2016, pp. 1–4.

[QPP02] X. Qie, R. Pang, and L. Peterson. "Defensive Programming: Using an Annotation Toolkit to Build DoS-resistant Software." In: *Proceedings of the 5th Symposium on Operating Systems Design and implementation*. OSDI '02. Boston, Massachusetts: USENIX Association, 2002, pp. 45–60.

[RM12] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014.

[SKM19] F. Seidel, K. Krentz, and C. Meinel. "Deep En-Route Filtering of Constrained Application Protocol (CoAP) Messages on 6LoWPAN Border Routers." In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. Apr. 2019, pp. 201–206.

[TGS17] M. Tiloca, C. Gehrmann, and L. Seitz. "On improving resistance to Denial of Service and key provisioning scalability of the DTLS handshake." In: *International Journal of Information Security* 16.2 (Apr. 2017).

[THA18] M. Tiloca, R. Hoglund, and S. Al Atiiq. "SARDOS: Self-Adaptive Reaction Against Denial of Service in the Internet of Things." In: *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*. Oct. 2018, pp. 54–61.

[WJS07] H. Wang, C. Jin, and K. G. Shin. "Defense Against Spoofed IP Traffic Using Hop-Count Filtering." In: *IEEE/ACM Transactions on Networking* 15.1 (Feb. 2007), pp. 40–53.

[XR03] Xiaofeng Wang and M. K. Reiter. "Defending against denial-of-service attacks with puzzle auctions." In: *2003 Symposium on Security and Privacy, 2003.* May 2003, pp. 78–92.

[Zol19] Zolertia. *Zolertia Firefly*. 2019. URL: `https://zolertia.io/product/firefly/`.

X-Pro: Distributed XDP Proxies against Botnets of Things

Abstract

The steadily increasing Internet of Things (IoT) devices are vulnerable to be used as bots to launch distributed-denial-of-service (DDoS) attacks. In this paper, we present X-Pro, a distributed XDP proxy to counteract DDoS attacks. We propose a source-based defense mechanism where proxies located between the IoT devices and the victim performs flow policing on all IoT traffic from a single administrative domain. The proposed proxy architecture can be integrated in widely used IoT frameworks as well as telecommunication networks. The proxies are working synchronously to block bogus messages and to detect traffic levels above predefined thresholds. Our implementation leverages eXpress Data Path (XDP), a programmable packet processing in the Linux kernel, as the main engine in the proxy. We evaluate X-Pro from several standpoints and conclude that our solution offers efficient DoS traffic blocking for both low-rate or massive attacks. Depending on the device side implementation selection, the computational overhead is cheap at the cost of some bandwidth loss.

Syafiq Al Atiiq, Christian Gehrmann. "X-Pro: Distributed XDP Proxies Against Botnets of Things". In 26^{th} *Nordic Conference on Secure IT Systems, Nordsec 2021, Tampere, Finland*. LNCS Vol. 13115, pp. 51–71, Springer.

1 Introduction

Denial of Service Attacks (DoS) has harmed the internet since the early 1980s. DoS prevents legitimate users from reaching their services and still constitute a major problem. In September 2016, a tremendous Distributed DoS (DDoS) attack was launched against several high-profile websites: OVH [Kla16], Dyn [Hil16], and Krebs on Security [Kre16]. Surprisingly, the source came from a vast amount of embedded devices turned into bots. A master process controls these bots, which is later known as the Mirai botnets [Kol+17]. Mirai scanned the internet and infected embedded devices running with insecure default password.

Based on the attack strategy, an adversary can design the botnets to be launched, either as a periodically low-rate [Luo+14; MDG09; Gui+05], or a massive [Sha+15b] DDoS attack. The low-rate DDoS behaves identically to the regular traffic pattern, making it difficult to detect by firewalls, routers, and switches. Furthermore, in an IoT setting, low-rate DDoS is not only a threat against the target nodes; as IoT devices often are battery-driven and resource-constrained, such attacks can severely harm the device itself.

Traditionally, DoS is handled at the victim-end or core-end, applying network-level detection and filtering [SA21]. However, as the type of DoS threats are very diverse, so are the suggested countermeasures. The majority of the works focus on the detection and blocking of harmful traffic [Mah+17]. Three main detection approaches occur in the literature: Signature-Based Approach (SBA), Anomaly Based Approach (ABA), and Entropy-Based Approach (EBA). SBA and ABA share many characteristics with traditional intrusion detection systems, while EBA is a pure traffic analysis approach [WYL10].

We have observed that this traditional way of handling DDoS does not consider the new IoT communication patterns. Especially, the following aspects are fundamentally different from an IoT perspective:

- IoT devices typically do not primarily communicate with general internet services but are directed towards a specific backend.
- Botnet threats on IoT entities are undesirable from a resource perspective, and this gives a large incentive for IoT device owners to implement DDoS countermeasures at the *source* not just on the network level.

Inspired by earlier successful of source-end approaches such as D-WARD [MR05], we reconsider the DDoS problem from an IoT application perspective. We argue that DDoS blocking and detection can efficiently be applied by strong policing on IoT *flow* level and that such policing preferably takes place at the IoT backend. As we show in our paper, X-Pro can be easily integrated into an existing IoT backend as well as into already deployed telecommunication network. To show that this is indeed an efficient approach, we have designed a DDoS filtering architecture based on simple flow counts where packet count and policing take

place on ordinary backend servers. This is very similar to an EBA detection mechanism, but we argue that we can filter using absolute flow thresholds in a strong and controlled IoT environment. To be able to get a fast packet processing in the kernel context, we utilize XDP [Høi+18]. X-Pro can be combined with traditional DoS detection mechanisms using, for instance, SBA or ABA. However, to secure basic functionality and IoT availability, the first step is to filter using flow thresholds. In this paper, we show how to use our architecture with such flow threshold values and policing. We call our solution X-Pro, coordinated XDP proxies running together as distributed systems to detect and filter out attack messages. Our solution can be combined with more advanced detection mechanisms, which will be left for future work. Our main purpose is to show that our approach is efficient in terms of overload blocking and that it can be implemented in existing IoT backends and devices with a reasonable performance overhead. In summary, the paper presents the following contributions:

- We provide a new framework (namely X-Pro) to counteract an attack towards a victim from the context of the IoT unit as the adversary.
- We suggest novel algorithms for packet filtering within a proxy and packet data synchronization between proxies.
- We provide an implementation and performance evaluation of the proxy using XDP, a novel programmable packet processing hook in the Linux kernel.
- We present IoT side realization and show the packet handling overhead.

The rest of the paper is organized as follows. We discuss related works and backgrounds in Sections II and III. We provide the solution in Section IV. Section V presents the implementation, while in Section VI, we provide the performance evaluation. Section VII draws our conclusions and anticipates future works.

2 Related Work

First, we distinguish between high and low-rate DDoS. We start by discussing the former and then continue with the latter. DDoS can be performed by flooding the victim with massive and bogus messages to consume bandwidth or resources. Such mechanism, behavior, mitigation strategies, and the detailed taxonomy are defined in [Sha+15b]. The author distinguishes the DDoS mitigation strategy into two categories, namely: (i.) *collaborative*, where multiple nodes are cooperating to mitigate DDoS, and (ii.) *non-cooperative*, in which no collaboration between network elements occurred.

Examples of *collaborative* strategies: FireCol [FAB12], CoFence [RFB17], and CoDef [LKG13]. FireCol observes the occurrence of DDoS attacks at the Internet Service Provider (ISP) level, in which the ISPs form a subscription mechanism

between each other. If FireCol detects an attack within an ISP, it informs the occurrence to the upstream ISPs, which consecutively perform a similar mitigation process. To detect such an attack, one of the most widely used method is Kullback-Leibler divergence and Shannon's entropy, where malicious traffic is detected based on IP address or packet size distribution statistics [Yu+11][XLZ11]. Our investigations use simple packet frequency thresholds for DDoS detection, and we also adopt a detection window approach. However, we do not focus on advanced detection rules but use firm and a priori thresholds. This is motivated by the fact that the main research goal in this paper *is not* to evaluate detection principles but rather to introduce a new proxy model for DDoS prevention, an efficient filter based on XDP and corresponding packet distribution data sharing principle between the proxies. Our approach can be extended to handle traditional, statistical detection methods based on the frequency counts provided by our solution.

Non-cooperative strategies include dynamic resource scaling [Yu+14], scaling via low-cost untrusted CDN (Content Delivery Network) [Gil+16], and harnessing DPI (Deep Packet Inspection) in SDN (Software Defined Networks) [Tsa+17]. The dynamic resource scaling offers a resource allocation strategy based on the queueing theory when idle VMs (Virtual Machine) are utilized once DDoS occurred. However, as the VM is not free, the occurrence of DDoS might emerge to become an Economic-DoS [ITJ13], where the adversary shifts the target to the economic aspect of the victim. In this case, the resources in the intermediate nodes are occupied as the attack reaches the network between adversaries and victims.

Both *collaborative* and *non-cooperative* mitigation approaches mentioned are for high-rate DDoS. We argue that the occurrence of high-rate DDoS should be solved along with the low-rate DDoS at the same time. Examples of low-rate DDoS are shrew-attack [Luo+14], LoRDAS [MDG09], and reduction-of-quality (ROQ) [Gui+05]. Shrew-attack abuses the weakness of TCP's retransmission timeout (RTO) procedure. A legitimate TCP flow is being attacked by regularly dispatching a high-rate bogus message simultaneously of the RTO. This way, once a sender restores from timeout, they will instantly receive a subsequent attack and probably go back to the timeout phase again. Low-rate DDoS are usually detected by employing a frequency domain analysis, i.e., Discrete-Fourier-Transform in one of the components in the system. Example of such mechanism is described in [FKA16]. To this point, X-Pro does not utilize or employ such analysis or any related statistical method to perform DDoS occurrence detection. But rather, we provide the data to be used by the system designer to employ such a method. It is possible to use an advanced method, i.e., machine learning working together alongside X-Pro, to perform more sophisticated analysis to set the X-Pro filtering thresholds. Extending X-Pro in this regard is for future work.

We utilize XDP extensively in this work. XDP has been around for a couple of years and has been used by some companies and open source projects to perform high-speed packet processing. Cloudflare [Ber17] publicly announce that XDP is

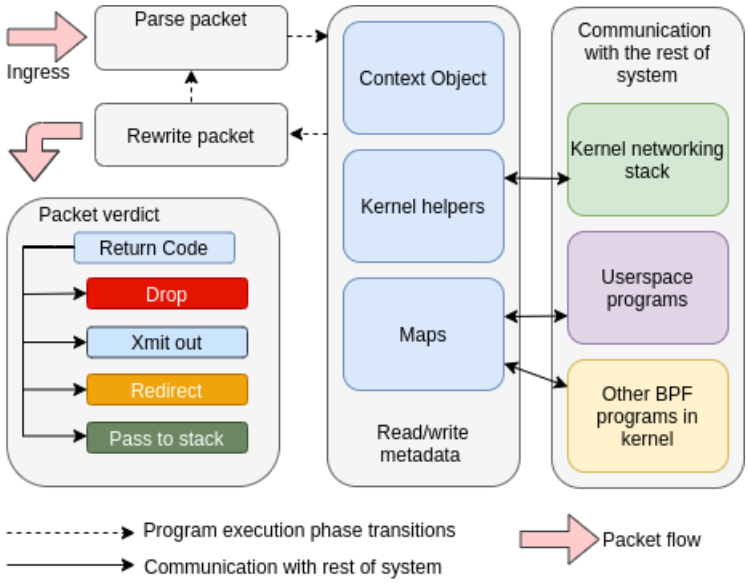

Figure 1: Execution flow of a typical XDP program, as described in [Høi+18]

used in their DDoS mitigation pipeline. Suricata, an open-source Intrusion Detection System (IDS) provides XDP plugin [Sur19] to their IDS. Also, Facebook has been extensively harnessing XDP as they claim that every packet reaching their network is being processing by XDP enabled application [Shi18].

3 XDP and BPF Maps

XDP [Høi+18] is a novel programmable packet processing hook living inside the kernel-space. In XDP, the underlying operating system accommodates a safe execution environment to run an eBPF [1] program. This execution happens within the device driver context. XDP has been part of the mainland Linux kernel since 4.8 [Mil16]. XDP provides a safe, fast, and customizable packet processing integrated with the kernel networking stack. An execution flow of a typical XDP program can be seen in figure 1. The logic of the eBPF program running inside the XDP hook is written in a high-level language, i.e., C, and compiled into bytecode. The kernel has the job of safeguarding the eBPF program by verifying it. This verification happens during the load time of the program.

Within the XDP execution environment, eBPF programs are executed in return to an event, i.e., when the packet arrives. The eBPF program does not have access to persistent memory within the boundary of its program context. Instead, the kernel provides the eBPF program a similar feature with access to a so-called

[1] eBPF [Fle17] refers to extended-BPF, the newer version of original BPF (Berkeley Packet Filter).

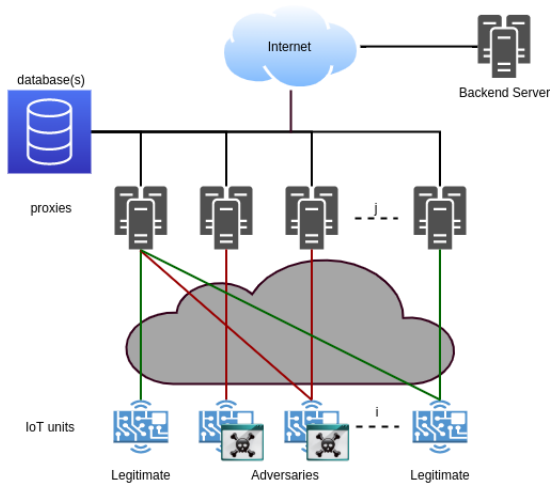

Figure 2: Overall X-Pro Architecture

BPF maps [Høi+18]. It is a generic data structure to store many different data types. Similar to a database, the format of BPF maps are key/value stores defined before loading an eBPF program to an interface. An eBPF code can refer to BPF maps within its codebase, just like referring to a memory. Fig 1 shows the relationship between BPF maps with other entities in the system.

4 The X-Pro Solution

In this section, we present X-Pro, a distributed XDP-based proxy to counteract DDoS. Under the assumption that an adversary can infect IoT units but not prevent packets from flowing over the proxies (see also our discussion regarding device-side implementation in Section 4.5), X-Pro prevents servers from being overwhelmed by DDoS by blocking the unwanted traffic with distributed proxies. When a filtering decision is taken, the system also automatically detects a potential DDoS attempt. To handle distributed DoS, information about traffic condition is shared between the proxies via a centralized database node. For the rest of the paper, we follow notations from table 1.

4.1 Overall Architecture and Solution

The X-Pro overall architecture can be seen in Figure 2. The first part is a set of proxy $P \ni p_1, ..., p_n$ interconnected through internal network. To have a broader view of the system at a particular time, each proxy node shares logging and load information between each other, leveraging a centralized database M_{DB}. The synched information is described in more detail in section 4.3. The next part in

U	Set of IoT units in the system
$u \in U$	An IoT unit
P	Set of proxies in the system
$p \in P$	Proxy
i	A unique index given to an IoT unit
u_i	An IoT unit with index i
j	A proxy unique network address associated with a proxy
p_j	A proxy with address j
M_{DB}	A centralized database
L_{DB}	A local database at p_j
D_{addr}	Destination IP address of a packet
ts'_1	Time stamp in p_j indicating the "oldest" packet time for a particular (i, D_{addr})
ts'_2	Time stamp in p_j indicating the "most recent" packet time for a particular (i, D_{addr})
ts_1	Time stamp in M_{DB} indicating the "oldest" packet time for a particular (i, D_{addr})
ts_2	Time stamp in M_{DB} indicating the "most recent" packet time for a particular (i, D_{addr})
ts_1*	The minimum value of ts_1 in p_j for a specific destination address among U
ts_2*	The maximum value of ts_2 in p_j for a specific destination address among U
c	Packet counter for a particular (i, D_{addr}) pair
dc	Delta packet counter (internally within a proxy) for a particular (i, D_{addr}) pair
$c*$	The sum of c and dc for a particular D_{addr}
T_{T1}	First filtering reset threshold used by a proxy
T_{T2}	Second Filtering minimum measure time threshold used by a proxy
T_{T3}	Third filtering minimum measure time threshold used by a proxy
T_{T4}	Fourth filtering minimum measure time threshold used by a proxy
T_{F1}	First packet maximum allowed frequency threshold used by a proxy
T_{F2}	Second packet maximum allowed frequency threshold used by a proxy
r	Frequency division factor

Table 1: Notations

the system is IoT units ($U \ni u_1, ..., u_n$). Each unit must have the connectivity to at least one available proxy within $p_1, ..., p_n$. It is possible to have multiple connections from a single u_i towards multiple p_j. We allow the adversaries to control IoT units U, meaning that a message proxy p_j receives from an IoT unit might be bogus.

One can apply X-Pro into an existing IoT backend, such as Thingsboard [Thi20] and Mainflux [Mai20]. In Thingsboard, there is a transport layer where the job is to receive messages from the devices, then parse the messages, which are then forwarded to one of the queues. X-Pro acts as a complement of Thingsboard transport with an additional feature of DDoS mitigation. As we show later in section 4.5, X-Pro requires an IoT device to set up an IP tunnel to the proxy. This fits naturally in the connection between the device and the Thingsboard transport. An advantage of this approach is the device does not have to resolve the domain of the Thingsboard transport as everything happens in the IP layer. However, as we require all the traffic from the device to pass the proxy, there is an additional mechanism to forward the packet outside the Thingsboard core if devices want to send the packet outside the X-Pro network. Hence, X-Pro can live to coexist without having to break the existing implementation.

Another option is to put X-Pro into an existing telecom infrastructure. The 5G network allows us to have multi-access edge computing (MEC) [Kek+18] close to the radio base station. As the purpose of MEC is to get an application closer to the user, we can utilize distributed MEC to be attached with X-Pro software in its network interface. This will make sure that the changes in the network operator side will be as minimum as possible. Even if, for example, 5G network deployment is still far away plan for some network operator, we can deploy X-Pro (within MEC) in the existing 4G network [Giu+18]. One possibility is to make X-Pro acts as a user plane packet inspection in the S1U interface between SGW (Serving Gateway) and eNodeB.

4.2 Filtering Design

We use a filtering approach where each packet arrived at a particular proxy, is analyzed, and potentially blocked[2]. Such mechanism is performed as early as possible, i.e., in the XDP hook. The blocking decision is based on a set of threshold parameters (see also the notation in table 1). These can be *tuned* to get the right trade-off between security and false blocking decisions. We discuss and review different threshold parameter selections, performance, and DoS packet endpoint reaching rates in Section 6.

The filtering is done using a time window, defined by two-time values, ts_1 and ts_2 for each target address for any ongoing traffic flow at any p_j. Each p_j is

[2]Blocking automatically also implies a potential attack detection. Our solution can be adapted such that we use lower threshold values for detection than blocking decisions. For simplicity, we only consider a single blocking threshold.

assumed to keep corresponding time values and packet counts. The current *local* view of the time window is denoted by ts'_1 and ts'_2. The packet frequency for each target destination is calculated regularly and compared with the frequency thresholds T_{F1} and T_{F2}. The first frequency threshold T_{F1} indicates the maximum allowed packet rate from one particular IoT unit to one *specific* endpoint. The second threshold T_{F2} indicates the *accumulated* maximum allowed packet rate (from all connected IoT units) towards one specific endpoint. In a corner case, it might be possible for the legitimate u_i to be falsely blocked by p_j due to a circumstance when a single victim is being attacked with a low-rate traffic from many compromised u_i. This will make the accumulated traffic become higher than T_{F2}. To accommodate this case, we provide a non-policed flow to be attached to a specific source (i.e. legitimate u_i). However, as it is now being done manually, the proposal for performing this task automatically is left for future work. For simplicity, we only consider a system-wide and *common* threshold values, T_{F1} and T_{F2}. However, the very same principle can be applied to a system where *individual* threshold values are given to specific destination addresses or destination address ranges. The latter would be the case envisioned for most applications, but our simplification will not make any major difference when evaluating the effectiveness of our approach. The complete filtering algorithm (algorithm 5) is shown in Appendix B.

4.3 Synchronization Design

To have the same visibility on each proxy, one needs to have a synchronization function between them. This section explains in more detail such procedures. For every time period t, each proxy p_j performs the synchronization procedure with the centralized database, M_{DB}. p_j iterates through every (i, D_{addr}) received from M_{DB}, and for each pair of (i, D_{addr}), p_j looks up at the corresponding values in the local database, L_{DB}. If a newer data is available, each entity (either p_j or M_{DB}) will update each other. The rule of thumb here is that a newer data will always replace an older one. The synchronization between p_j and M_{DB} are assumed to be over a secure end-to-end communication channel. We consider $p \in P$ are trustworthy, hence a direct attack on them are out of scope of this paper. We have come up with a synchronization procedure that tries to cover the most important aspects for keeping consistent time window and counts among the proxies; see the algorithm 4 in Appendix A. The way M_{DB} implemented is agnostic to any particular technology. We assume that M_{DB} is robust, reliable, and impossible to be killed no matter how high the DDoS attack is. To achieve that, it is possible to design M_{DB} as a distributed system as described in [Agr+11] [Jim+02]. However, the design of such system is out of scope of this paper.

The main challenge with the synchronization lies in the counts and count window in L_{DB} might be different from M_{DB} since the last synchronization. Hence, the synchronization must be able to cope with these changes. The algorithm han-

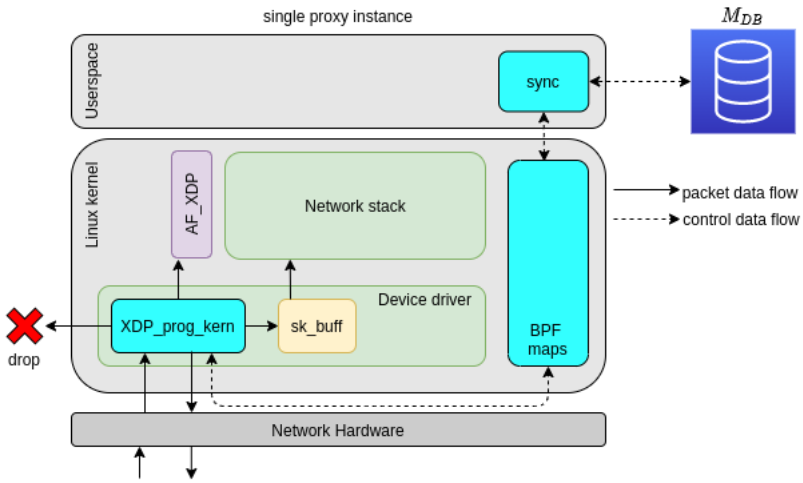

Figure 3: A proxy instance

dles this by comparing the local time window ts'_1 and ts'_2 with the corresponding ones loaded from M_{DB}, ts_1 and ts_2. Furthermore, to stop the count window growing to infinity, we must reduce the size if the time window is getting too large. Such identification happens locally, and the rest of the proxies must adjust their values accordingly at their next synchronization. Besides, if no record for a particular address occurs for a while, our filtering algorithm resets the count (this is indicated through a `mark` parameter). Such reset should be propagated to the rest of the proxies *only* if they have not received a similar destination packet for a time exceeding a predefined threshold. These aspects, as well as making sure that the counts and the window are consistent, are the primary purpose of our synchronization design which has been verified through the experiments. The different time thresholds can be tuned to get the suitable trade-off between filtering efficiency and synchronization overhead.

4.4 Proxy Design Based on XDP

As mentioned in section 3, XDP can process the packet at the earliest possible hook. An advantage of this feature is the associated device driver, which handles the packet, does not need to allocate the memory if it turns out that the incoming packet is not legitimate. This, in return, would require a lot less resource on the proxy if the attack turns out to be massive. We intend to exploit this feature as the underlying mechanism as defined in more detail in this chapter.

As described in 4.3, the information ts_1, ts_2, c for each incoming packet needs to be shared among P. It is natural to pass this information to p_j's userspace first, then perform the synchronization with M_{DB} from there. The sync function

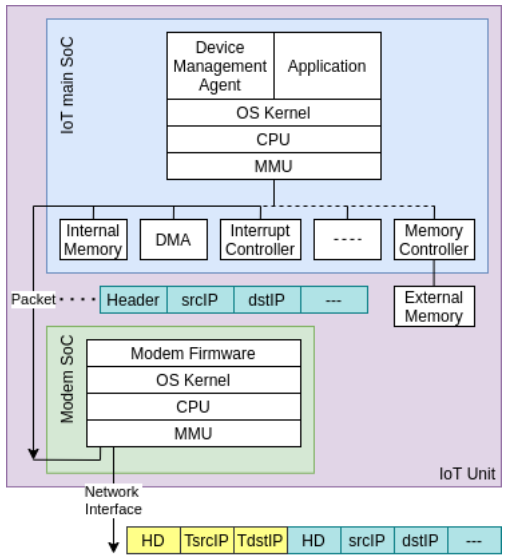

Figure 4: An IoT implementation

should be executed between the local database, L_{DB}, which is represented by BPF maps, and the centralized M_{DB}. It is fairly trivial to integrate such mechanism using API provided by M_{DB} as well as local API of BPF maps [Pag20]. For any new information ts_1, ts_2, c recorded from the XDP program, the data is stored in L_{DB}. If newer data is found, the old data is replaced.

Once the local BPF maps are filled with the needed information, the next job is to propagate this to all other proxies in P, via M_{DB}, as well as retrieving data from M_{DB} that is not available locally. Note that all the synchronized information is only statistics of the packet, not the packet itself. Meaning that, only small amount of transactions between p_j and M_{DB} is needed. This mechanism allows us to pass the information between proxies without sending the invalid packet to the userspace. Hence, reducing the resource utilization in the userspace. Therefore, the proxy can allocate the resource to a more essential task, i.e., packet filtering in the kernel. Our rationale is that the fewer tasks performed in the userspace, the more resource can be utilized by the XDP to block the invalid messages; hence we get more packet filtering capacity in the kernel. Furthermore, as our codebase does not require access to a specific kernel helper, X-Pro can be offloaded entirely to, i.e., smart-NIC [Mia+19] to get a better performance. Figure 3 represents the mechanism mentioned in this section.

4.5 Device Side Design

The X-Pro design requires *a*ll traffic from the IoT units are routed through the proxy. An attacker aware of this principle can circumvent this mechanism by avoiding the whole proxy network, and fulfill the DoS target. Therefore, it is a mandatory design requirement for the IoT device to prevent its IP traffic control part from being infected by a malicious software. A legacy IoT device can connect to p_j as long as it has a way to separate the main and network MCU securely, i.e. through secure virtualization. Several different techniques are possible. In this section, we discuss a possible design where the modem SoC is separated from the main SoC, allowing secure proxy packet encapsulation.

The job of the modem SoC is to perform proxy management and provides a standard network interface from the operating system within an IoT device. Figure 4 shows the connection between the main SoC and the modem SoC within the IoT unit u_i. The modem SoC keeps track of available proxies P in the system. It is possible to change the current destination proxy, p_j, if, for example, the one currently used is overloaded or unavailable. Information about load and availability is obtained through a probing mechanism. Two different solutions are possible:

- The modem SoC connected to an arbitrary proxy p and received the load information of all the proxies within P. The received information is then used to decide which p_j has the minimum load among the proxies, P.
- The modem SoC measures multiple p_j within P at once by sending a ping request and calculate the lowest response time among the measured proxies.

As shown in figure 4, the modem SoC embed an additional header as a tunnel header. It tunnels all outbound traffic to the selected proxy from the previous mechanism. The tunnel itself can be implemented as a raw IP tunnel, HTTP, or even CoAP [SHB14] where the source IP packet is encapsulated. This way, X-Pro would not prevent an IP level (or even HTTP) end-to-end security, such as IPSec. Unlike the outbound connections, the inbound traffic is treated entirely transparent and does not affect the modem or the whole IoT unit in any way.

5 Implementation

This section will describe the technical implementation of proxies, the centralized database, and the device packet handling in more detail. The code is available as open-source[3].

[3] https://github.com/syafiq/xpro

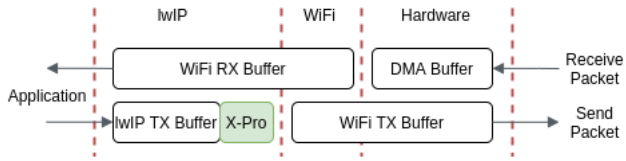

Figure 5: TCP/IP stack modification in ESP32

5.1 Proxy

The proxy implementation consists of two different parts, that is kernel space and userspace. The kernel space implementation mostly deals with the filtering mechanism for each incoming packet, whereas the userspace implementation deals with the synchronization between proxies. Within the kernel space, the algorithm 5 is implemented as an eBPF program [Tea20a], written in C. The eBPF program is attached to one of the interfaces in the proxy, p_j.

Each proxy p_j synchronizes the local BPF map to M_{DB} for every predefined period t. The synchronization utilizes BPF helpers [Tea20b], i.e. `bpf_map_lookup_elem` and `bpf_map_get_next_key`. These functions is periodically called to iterate through the BPF maps and update the M_{DB}. To read-/write through the M_{DB}, we harness hiredis [SNR20], a redis client written as a C library.

5.2 Centralized Database

The database M_{DB} is a single Redis instance running inside a virtual machine. M_{DB} must have connectivity to all the proxies in the system.

5.3 IoT units

We have developed the proof of concept for the IoT units using a low-cost platform ESP32 [Esp20] and FiPy [Pyc20]. The aim is to provide a real-world example of performing procedures explained in section 4.5. The process should be transparent to the applications running in the main SoC, and both processes should be completely separated from each other. It means the application should not handle the proxy selections and packet encapsulation mechanism, but rather the modem SoC does. This gives a solid separation between the systems and strong protection against software attacks of the main system. In our proof of concept, the ESP32 acts as the modem SoC, whereas FiPy acts as the main SoC. Both boards are connected through the UART pinouts. In our PoC, the ESP32 board runs on a native operating system from espressif, esp-idf [Esp20]. We modify the firmware such that any outgoing packet is always encapsulated, with one of the proxy p being the new destination IP. As mentioned earlier, the old destination IP is preserved, along with the payload from the application in the main SoC.

As UART speed is fairly slow compared to, i.e., wireless connection, we decided to build a second prototype in which the encapsulation process happens in the TCP/IP stack. We modify the lightweight TCP/IP stack from the ESP32 firmware at the last point of IP encapsulation (within the lwIP TX buffer) before the packet is moved into the WiFi TX buffer. Figure 5 shows where exactly our modification happens within the ESP32 TCP/IP stack. From a security point of view, this would require either of the two following options:

- The logic from section 4.5 is implemented in a hardware (i.e., VHDL), such that adversaries cannot tamper or modify the encapsulation process.
- The TCP/IP stack lives in a trusted environment (i.e., ARM TrustZone[Arm20][PS19]) such that the isolation is built-in into the main CPU and SoC.

In our proof of concept, we have *n*ot made a full implementation of any of these two options. However, the overhead with a pure hardware solution would undoubtedly be less than our chosen proof of concept implementation. The TrustZone option or any other virtualization options like using a thin I/O hypervisor [Shi+09] is left for future work.

6 Experimental Evaluation

Next, we evaluate the design and realization through a proof of concept. We have made a full implementation of the design on the proxy and the device side as described in Section 5. The proxies and the centralized database are implemented as a virtual machine in Fedora 30 operating system, running kernel version 5.6. All of the VMs of the proxies and centralized database are running with one vCPU and 1024 MB of memory. We simulate the IoT units (infected and non-infected with botnets) with a Linux machine, running PKTGEN [TSO16] software from the Linux kernel tree, with adjustable intensity. PKTGEN sends CoAP messages, in which the size of each packet is 64 bytes. Message rates can easily be set through the *ratep* value in the PKTGEN configuration file.

Our evaluation goal is to measure how effective the suggested solution in terms of packet blocking for both single high-rate attack and low-rate attacks using fixed attack thresholds. We also measure the pure overhead at the device side for our two different implementation options (see also Section 5.3). As some IoT units might be expected to send relatively high amount of traffic, typically directed to a particular server node, we would like to measure how well our DoS blocking principle works in a situation where we have a mixture of such high rate, valid traffic, and DoS traffic. We use a simple approach where some IoT units are allowed to send traffic at their maximum capacity without being blocked while the rest are subject for the filtering with thresholds. In a more realistic setting, the flows that should be policed or not can be set in a more fine-grained way and vary over time.

However, to simplify our measurement, we only use two static categories of devices, i.e., units with non-policed traffic and devices with policed flows. A more advanced principle that label flows in a more intelligent way using for instance, machine learning, is left for future work [BE18][Zhi+20].

6.1 Single-Proxy

In this scenario, a single proxy instance p_j sits between IoT units and the victim. An infected unit becomes an adversary and a unit without policed traffic (without upper threshold) that is assumed to be not infected by the botnets. Both the units, either the infected or the non-blocked one, are sending packets towards the victim with various intensities, ranging from 50000-400000 packets per second. These values are picked merely based on the capability of our test hardware[4]. While performing packet forwarding to the backend server (or victim), p_j drops incoming messages if it senses a DoS attack. This mechanism is based on the algorithm we mention in section 4.4. Even though it does not seem to make sense that a single IoT unit can generate such magnitude of the attack, we argue this measurement is still crucial for the following reason. We can test the limit on how high proxy p_j can cope with a DoS attack within the context of provided hardware, i.e., one vCPU and 1 GB RAM.

Figure 6 shows the percentages of packets being passed to the backend server by p_j varies to the attack intensity in packets per second (PPS). As the intensity of the incoming packet exceeds the value of TF_1, more than 99% of the packets are efficiently dropped if the source comes from the infected IoT units. When this happens, only less than 1% of the bogus messages are forwarded as p_j needs some time to calculate the frequency before deciding what action needs to be performed for the subsequent packet. For example, when $TF_1 = 300k$ and attack messages = $400k$, only 0.3% of total messages are forwarded to the final destination, while the rest are dropped. However, no messages from IoT units marked to not be subject to blocking are dropped by p_j.

6.2 Multiple-Proxy Working Together

In this scenario, a set of proxies $p_j \in P$, where $j = 1, 2, 3, 4$ sits between multiple IoT units. Also, a set of IoT units $u_i \in U$ where $i = 1, 2, ..., 8$ are connected to the set of proxies P. Each p_j is connected to two IoT units $u_{i_{odd}}$ and $u_{i_{even}}$, where $u_{i_{odd}} = 2j - 1$ and $u_{i_{even}} = 2j$. It means, for example, p_1 is connected to u_1 and u_2, p_2 is connected to u_3 and u_4, and so on. In this experiment, we have set $u_{i_{odd}}$ to act as an infected IoT unit, while $u_{i_{even}}$ are not subject to traffic policing, i.e., they are allowed to send traffic at their maximum capacity. Each unit sends

[4]To put into perspective, the whole New York area has been deployed with 15000 security cameras by NYPD[Fus21]. So, depends on what application is used and how many sensors needed, we think that 50000-400000 make sense.

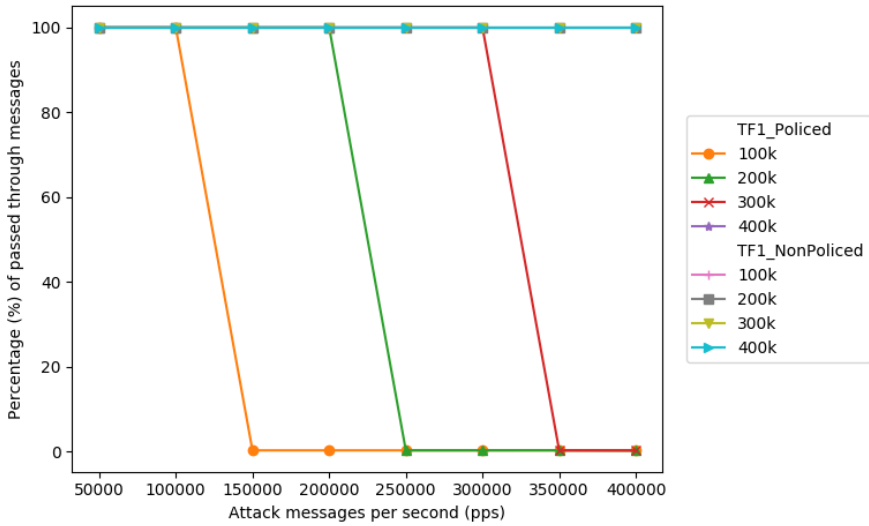

Figure 6: % of passed-through messages on a single proxy varies to the attack intensity

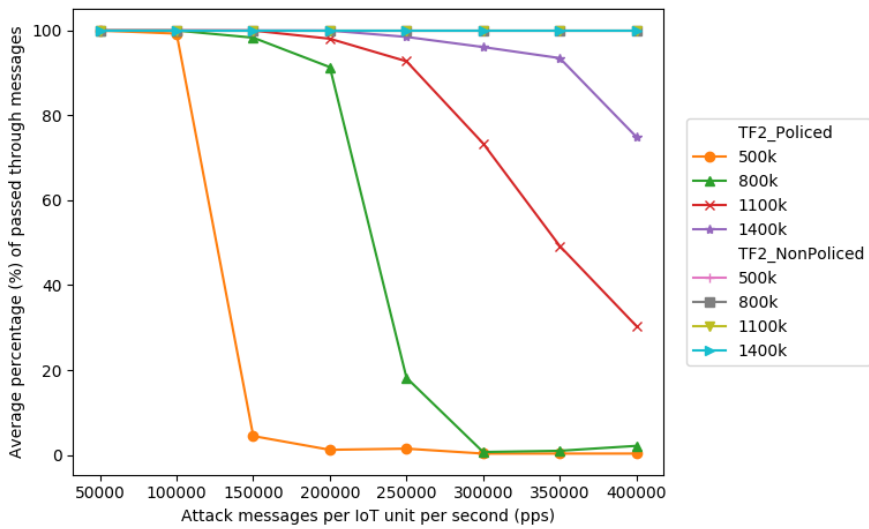

Figure 7: Average % of passed through messages on the multiple proxies P varies to the attack intensity

	Separated main MCU and network MCU	lwIP logic of encapsulation	
	Average RTT (ms)	Average RTT (ms)	Average throughput (Mbits/s)
X-Pro	50.55	37.67	25.86
vanilla	35.96	36.78	30.37

Table 2: Overhead of the IoT Device

packets to the victim with various intensities, ranging from 50000-400000 PPS, and p_j performs a synchronization towards M_{DB} for every 4 seconds. Note that these values are determined heuristically, in which the equation and/or derivation of such values are out of the scope of this paper.

We perform this measurement to show how X-Pro handles a situation where multiple adversaries are trying to launch an attack under the radar, i.e., sending attack messages just below TF_1 with the hope that attack packets can slip through p_j. As we can see later, this is not the case as p_j has been implemented with the procedure mentioned in section 4.4. Figure 7 shows the average percentages of packets being passed by p_j vary to the packet intensity per second from each u_i. In the infected IoT units $u_{i_{odd}}$, we can clearly see that all $p_j \in P$ start to drop incoming packet from $u_{i_{odd}}$ when $4\times$ incoming packets is greater than TF_2. This is expected, as we have four units of devices turned into botnets.

Among the units $u_{i_{even}}$, all packets are forwarded to the backend server as the incoming packets from these sources are not counted to get the frequency value. We can see from figure 7 that the average percentages for $u_{i_{even}}$ are 100%.

6.3 Overhead from the IoT units

First, we measure our prototype where the main and the network MCU are completely separated hardware connected through UART. The main MCU sends a CoAP request to the network MCU, followed by stripping off the old destination IP with p_j as the new destination address. It is then sent to p_j and forwarded to the final destination if deemed as a non-malignant packet. When the network MCU receives the CoAP response, it is delivered back to the main MCU. All those processes are counted while we measure the round trip time of the response-request messages. We measure 100 times and calculate the average, shown in table 2. It is clear that X-Pro requires an additional 15 ms processing time. The main reason for this overhead is the relatively slow UART communication. We did not measure the throughput of this approach as the UART will permanently cap it. However, the adversary can't tamper with the network MCU by having two different entities in a separate hardware.

Second, to get a clear picture of how much overhead is added when we have such an encapsulation process, we decided to implement our solution in a fully

	Average RTT (ms)
Thingsboard + X-Pro	82.93
Thingsboard	80.98

Table 3: Thingsboard with and without X-Pro

softwarized manner. However, this is with the assumption that already mentioned in section 5.3. We can see from table 2 that our solution only adds about 1 ms of the total round trip time. However, the throughput is a bit decreased by around 5Mbits/sec. The reason is there is additional overhead to process each packet due to the IP tunnel, i.e., encapsulating a new header such that the new destination IP is one of the proxy $p_j \in P$. This gives an extra 32 bits (the size of an IPv4) for each packet because the real destination should be preserved while the new destination is installed.

All the IoT unit overheads and extra implementation penalties can be avoided in the network situations where layer two mechanisms allow network-enforced routing through proxies. It is possible to mix such configuration with IoT IP tunneling configurations for some IoT units in the system. It is also possible to dynamically switch on and off the proxy forwarding function in the IoT units. When a DDoS attack is not expected, the protection mechanism is switched off, avoiding the bandwidth loss penalty for a certain amount of time.

To complement the example that has been provided in section 4.1, we measure the average round trip time (RTT) from the IoT device to the Thingsboard backend. The IoT device sends a CoAP message with the method POST and subsequently expecting an acknowledgment message. We measure the RTT as the time difference when the IoT device sends the CoAP message and receives the respective acknowledgment. The CoAP POST message is repeated 100 times, and the average value is calculated, as shown in table 3. We can see that X-Pro adds 2 ms overhead, a small fraction of the total average time.

7 Conclusion and Future Work

This paper has presented X-Pro, a distributed XDP proxies against botnets of things. The design of distributed proxies is armed with a centralized database, allowing the proxies to inform each other about the latest event in the networks. Through this collaboration, it is possible to defend the victim amid the situations when the adversaries are trying to send, (i.) a massive and well-coordinated attack towards the victim through a single proxy, (ii.) periodically low-rate bogus messages spanned to multiple proxies in which the intention is to fly under the radar. We have proven that this is indeed the case in practice through our experimental evaluations. The obtained results show that our solution allows strong protection of overload to both of the IoT backend and external attack targets. X-Pro requires

the IoT units to be modified in regards to the network interface modem. As the adversary cannot tamper with the modification, they cannot re-route the destination of an outgoing packet, which always be forwarded to one of X-Pro's proxies.

Our paper shows that X-Pro is possible to realize with low overhead in typical IoT scenarios and that it can be used to give protection from all kinds of packet overload attacks. In future work, we will extend the solution with more advanced DDoS detection mechanisms (i.e., machine learning), which will allow automatic DDoS infection detection combined with efficient blocking.

Acknowledgements

Work supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research and the EU H2020 project CloudiFacturing under grant 768892.

Appendix A Proxy Synchronization Protocol

Algorithm 4 Proxy Synchronization Protocol

p_j looks all the pair (i, D_{addr}), for $u_i \in U$
for each (i, D_{addr}) in M_{DB} **do**
  if $L_{DB} \ni i, D_{addr}$ **then**
    if $(mark' = 1)$ **then**
      $< mark' = 0 >$
      if $(ts'_1 - ts_1 > T_{T1})$ **then**
        $< dc' = 0 >$
        $< ts_1 = ts'_1, ts_2 = ts'_2, c = c' >$
      end if
    else if $(mark' \neq 1)$ **or** $(mark' = 1 \ \& \ ts'_1 - ts_1 < T_{T1})$ **then**
      if $ts'_2 < ts_2$ **then**
        $< ts'_2 = ts_2 >$
      else
        if $ts'_2 - ts_1 > T_{T4}$ **then**
          $< ts'_1 = ts'_2 - (ts'_2 - ts_1)/r >$
          $< c = \lfloor c/r \rfloor, ts_1 = ts'_1, ts_2 = ts'_2 >$
        end if
        $< ts_2 = ts'_2 >$
      end if
      if $ts'_1 > ts_1$ **then**
        $< ts'_1 = ts_1 >$
      else
        if $ts'_2 - ts'_1 > T_{T4}$ **then**
          $< ts'_1 = ts_1 >$
        else
          $< ts_1 = ts'_1 >$
        end if
      end if
      $< c' = c + dc', c = c', dc' = 0 >$
    end if
    $< c = c', dc' = 0 >$
  else
    $< ts'_1 = ts_1, ts'_2 = ts_2 >$
    $< c' = c, dc' = 0, mark' = 0 >$
  end if
end for
for each (i, D_{addr}) in L_{DB} **do**
  if $M_{DB} \ni i, D_{addr}$ **then**
    $< ts_1 = ts'_1, ts_2 = ts'_2, c = c' >$
  end if
end for

Appendix B Packet Filtering Procedures

Algorithm 5 Packet Filtering Procedures

<Lookup ts'_1, ts'_2, c for record (i, D_{addr}) in L_{DB} >
if record found **then**
    if $t - ts'_2 > T_{T1}$ **then**
        $ts'_1 = t, c' = 0, dc = 0, mark = 1$
    end if
else
    $ts'_1 = ts'_2 = t, c' = 0, dc = 0, mark = 0$
end if
$c' = c' + 1, dc = dc + 1, ts'_2 = t$
if $ts'_2 - ts'_1 > T_{T2}$ **then**
    if $c'/(ts'_2 - ts'_1) > T_{F1}$ **then**
        <Drop packet>
        <Send an overload warning>
    end if
end if
$ts_1* = \min_{u_i \in U_{D_{addr}}} ts_1'_i$
$ts_2* = \max_{u_i \in U_{D_{addr}}} ts_2'_i$
$c* = \Sigma_{u_i \in U_{D_{addr}}} (c_i + dc_i)$
if $ts_2 * - ts_1 * > T_{T3}$ **then**
    if $c * /(ts_2 * - ts_1 *) > T_{F2}$ **then**
        <Drop packet>
    end if
end if
<forward packet>

References

[Agr+11] D. Agrawal et al. “Database Scalability, Elasticity, and Autonomy in the Cloud.” In: *Database Systems for Advanced Applications*. Ed. by J. X. Yu, M. H. Kim, and R. Unland. Springer Berlin Heidelberg, 2011, pp. 2–15.

[Arm20] Arm. *Arm TrustZone Technology*. 2020.

[BE18] M. H. Bhuyan and E. Elmroth. “Multi-scale Low-Rate DDoS Attack Detection Using the Generalized Total Variation Metric.” In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, pp. 1040–1047.

[Ber17] G. Bertin. *XDP in practice: integrating XDP in our DDoS mitigation pipeline*. 2017.

[Esp20] Espressif. *ESP32*. 2020.

[FAB12] J. Francois, I. Aib, and R. Boutaba. “FireCol: A Collaborative Protection Network for the Detection of Flooding DDoS Attacks.” In: *ACM* (2012).

[FKA16] R. F. Fouladi, C. E. Kayatas, and E. Anarim. "Frequency based DDoS attack detection approach using naive Bayes classification." In: *2016 39th International Conference on Telecommunications and Signal Processing (TSP)*. 2016, pp. 104–107.

[Fle17] M. Fleming. *A thorough introduction to eBPF*. 2017. URL: https://lwn.net/Articles/740157/.

[Fus21] S. Fussell. *The All-Seeing Eyes of New York's 15,000 Surveillance Cameras*. 2021.

[Gil+16] Y. Gilad et al. "CDN-on-Demand: An affordable DDoS Defense via Untrusted Clouds." In: *NDSS*. 2016.

[Giu+18] F. Giust et al. "MEC deployments in 4G and evolution towards 5G." In: *ETSI White paper* 24.2018 (2018), pp. 1–24.

[Gui+05] M. Guirguis et al. "Reduction of quality (RoQ) attacks on Internet end-systems." In: *IEEE Infocom*. Vol. 2. 2005, pp. 1362–1372.

[Hil16] S. Hilton. *Dyn Analysis Summary Of Friday October 21 Attack*. 2016.

[Høi+18] T. Høiland-Jørgensen et al. "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel." In: CoNEXT '18. Heraklion, Greece: ACM, 2018, pp. 54–66.

[ITJ13] J. Idziorek, M. F. Tannian, and D. Jacobson. "The Insecurity of Cloud Utility Models." In: *IT Professional* 15.2 (2013), pp. 22–27.

[Jim+02] R. Jimenez-Peris et al. "Improving the scalability of fault-tolerant database clusters." In: *Proceedings 22nd International Conference on Distributed Computing Systems*. 2002, pp. 477–484.

[Kek+18] S. Kekki et al. "MEC in 5G networks." In: *ETSI white paper* 28 (2018), pp. 1–28.

[Kla16] O. Klaba. *Octave Klaba Twitter*. 2016.

[Kol+17] C. Kolias et al. "DDoS in the IoT: Mirai and Other Botnets." In: *Computer* 50.7 (2017), pp. 80–84.

[Kre16] B. Krebs. *KrebsOnSecurity Hit With Record DDoS*. 2016.

[LKG13] S. B. Lee, M. S. Kang, and V. D. Gligor. "CoDef: Collaborative Defense against Large-Scale Link-Flooding Attacks." In: CoNEXT '13. Santa Barbara, California, USA: ACM, 2013, pp. 417–428.

[Luo+14] J. Luo et al. "On a Mathematical Model for Low-Rate Shrew DDoS." In: *IEEE Transactions on Information Forensics and Security* 9.7 (2014), pp. 1069–1083.

[Mah+17] T. Mahjabin et al. “A survey of distributed denial-of-service attack, prevention, and mitigation techniques.” In: *International Journal of Distributed Sensor Networks* 13.12 (2017).

[Mai20] Mainflux. *Open Source IoT Platform*. 2020.

[MDG09] G. Macia-Fernandez, J. E. Diaz-Verdejo, and P. Garcia-Teodoro. “Mathematical Model for Low-Rate DoS Attacks Against Application Servers.” In: *IEEE Transactions on Information Forensics and Security* 4.3 (2009), pp. 519–529.

[Mia+19] S. Miano et al. “Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case.” In: *IEEE Access* 7 (2019), pp. 107161–107170.

[Mil16] D. Miller. *[GIT] Networking*. 2016.

[MR05] J. Mirkovic and P. Reiher. “D-WARD: a source-end defense against flooding denial-of-service attacks.” In: *IEEE Transactions on Dependable and Secure Computing* 2.3 (July 2005), pp. 216–232.

[Pag20] L. M. Page. *bpf - perform a command on an extended BPF map or program*. 2020.

[PS19] S. Pinto and N. Santos. “Demystifying Arm TrustZone: A Comprehensive Survey.” In: *ACM Comput. Surv.* 51.6 (Jan. 2019).

[Pyc20] Pycom. *Fipy*. 2020.

[RFB17] B. Rashidi, C. Fung, and E. Bertino. “A Collaborative DDoS Defence Framework Using Network Function Virtualization.” In: *IEEE Transactions on Information Forensics and Security* 12.10 (2017), pp. 2483–2497.

[SA21] M. Sharma and B. Arora. “Detection and Prevention of DoS and DDoS in IoT.” In: *Proceedings of Second International Conference on Computing, Communications, and Cyber-Security*. Ed. by P. K. Singh et al. Singapore: Springer Singapore, 2021, pp. 845–855.

[Sha+15b] A. Shameli-Sendi et al. “Taxonomy of Distributed Denial of Service mitigation approaches for cloud computing.” In: *Journal of Network and Computer Applications* 58 (2015), pp. 165–179.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014.

[Shi+09] T. Shinagawa et al. “BitVisor: A Thin Hypervisor for Enforcing i/o Device Security.” In: VEE ’09. Washington, DC, USA: ACM, 2009, pp. 121–130.

[Shi18] N. V. Shirokov. *XDP: 1.5 years in production. Evolution and lessons learned.* 2018.

[SNR20] S. Sanfilippo, P. Noordhuis, and J. Rediger. *Hiredis*. 2020.

[Sur19] Suricata. *eBPF and XDP*. 2019.

[Tea20a] L. K. Team. *BPF Documentation*. The Linux Kernel documentation. Aug. 2020.

[Tea20b] L. K. Team. *BPF-HELPERS*. 2020.

[Thi20] Thingsboard. *Thingsboard*. 2020.

[Tsa+17] S.-C. Tsai et al. "Defending cloud computing environment against the challenge of DDoS attacks based on software defined network." In: *Advances in Intelligent Information Hiding and Multimedia Signal Processing*. Springer, 2017, pp. 285–292.

[TSO16] D. Turull, P. Sjödin, and R. Olsson. "Pktgen: Measuring performance on high speed networks." In: *Computer Communications* 82 (2016), pp. 39–48.

[WYL10] J. Wang, X. Yang, and K. Long. "A new relative entropy based app-DDoS detection method." In: *The IEEE symposium on Computers and Communications*. 2010.

[XLZ11] Y. Xiang, K. Li, and W. Zhou. "Low-Rate DDoS Attacks Detection and Traceback by Using New Information Metrics." In: *IEEE Transactions on Information Forensics and Security* 6.2 (2011), pp. 426–437.

[Yu+11] S. Yu et al. "Traceback of DDoS Attacks Using Entropy Variations." In: *IEEE Transactions on Parallel and Distributed Systems* 22.3 (2011).

[Yu+14] S. Yu et al. "Can We Beat DDoS Attacks in Clouds?" In: *IEEE Transactions on Parallel and Distributed Systems* 25.9 (2014), pp. 2245–2254.

[Zhi+20] W. Zhijun et al. "Low-Rate DDoS Attack Detection Based on Factorization Machine in Software Defined Network." In: (2020).

Regaining Dominance in CIDER and Lazarus

Paper III

Abstract

Ensuring availability is a critical requirement for the Internet of Things (IoT). CIDER, a recovery architecture, and its follow-up scheme, Lazarus, are solutions to address this issue. CIDER introduced a new hardware module, the Authenticated Watchdog Timer (AWDT), to keep IoT devices running in normal mode as long as trusted authenticated tickets are received from a hub. If valid tickets are not received, the AWDT resets the device, and a recovery procedure takes over. Lazarus, a more advanced solution, implemented the AWDT in ARM TrustZone (TZ). CIDER and Lazarus promised to **unconditionally** recover control in case of compromised firmware. In this work, we analyze both and demonstrate that the schemes do not give such unconditional recovery. In particular, we identify two major problems. Attackers can easily make devices unavailable by manipulating the tickets. Even more severe is the fact that a runtime attack, utilizing a potential weakness in the trusted firmware, can also make a device unavailable and prevent the recovery mechanism. We notice that when the AWDT is implemented in TZ, a richer device security state information can be tracked securely, allowing the device to handle attacks in a much more robust way with a better availability guarantee. We design and implement a new solution using the same hardware as Lazarus. Our design introduces a new boot mode called LZ_VERIFY, which verifies whether disruption is caused by a network issue or a runtime attack, enhancing the device's decision-making for subsequent actions. Assuming the time spent in LZ_VERIFY as uptime, our evaluation shows that under network attack

Syafiq Al Atiiq, Christian Gehrmann. "Regaining Dominance in CIDER and Lazarus". In *IEEE Access, 2024*. vol. 12, pp. 124589-124603, IEEE.

conditions, the new design improves device availability from 1.66% to 99.95% compared to Lazarus, a significant improvement. A similar design is also possible using a hardware AWDT, as used by CIDER.

1 Introduction

The growth of IoT devices has been rapidly increasing and is expected to continue to rise. As of 2023, the number of connected IoT devices has already surpassed 15.7 billion and is projected to reach 38.9 billion by 2029[1]. IoT has been utilized in a diverse range of fields, i.e., building automation, healthcare, and energy. The various applications translate to a vast array of deployments with distinct requirements. Many of them are carried out on a constrained platform and rely on wireless connectivity.

It is crucial that applications are tailored to minimize power consumption and memory usage. IoT devices are typically deployed in remote locations where they are difficult to access and maintain (they often have minimal user interfaces). As a result, IoT management solutions must be designed to be secure, robust, and remotely maintainable at the same time. The distributed nature, constrained, and large-scale deployments make IoT devices an attractive target for adversaries.

IoT security is a complex field that requires careful consideration of various aspects. A comprehensive analysis of the complexities, layers, and threats is provided in [Wil+22]. Since attacks can target individual or multiple layers simultaneously, predicting and preventing them can be challenging. Therefore, taking a network-centric approach to IoT security is insufficient; it is also necessary to consider the application layer and physical attacks.

To enhance IoT security, it is essential to have strict control of the device software state, particularly since many devices are deployed with identical software and configurations. Attestation is a solution that enables the tracking of the software status, as explained in [Gro19; Sai+04; AC15]. While attestation is a powerful tool, it only enables a trusted entity to check/verify the software state, and additional measures are needed to ensure device availability.

CIDER [Xu+19] introduced the concept of *dominance*, allowing strict control of the software state of an IoT unit through a combination of software attestation and the usage of a new hardware primitive called AWDT. It keeps the device running in normal mode as long as authenticated tickets are received from a trusted hub, resetting the device and activating a recovery procedure if valid tickets are not received.

While CIDER implements this concept through a new IoT hardware module, Lazarus [Hub+20] reconsiders CIDER using off-the-shelf ARM hardware and TZ

[1] `https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/iot-connections-outlook`

to minimize the Trusted Computing Base (TCB). Both claim to have unconditional recovery even after a firmware compromise.

This paper examines the designs of CIDER and Lazarus, which rely on similar principles. We argue that both do not live up to the promise of *unconditional* recovery, at least in two cases we have observed.

First, a simple message tampering attack (which is explicitly allowed in the threat model of both Lazarus and CIDER) between the hub and the device causes the device to go into a cyclic reboot every AWDT period.

Second, a runtime attack from the application that affects the availability of the AWDT message triggers the same reboot (not every AWDT period, but every time the runtime attack is triggered). The reason for this is the limited recovery state handling that makes it difficult for the device and hub to distinguish between network-based and compromised firmware runtime attacks. These limitations pose a significant risk, as network operators can take action to address the former, while the latter can be resolved through a software upgrade or by changing the device's mode of operation.

To address these issues, we propose a new design that utilizes the small TCB to enable more robust state-tracking. This enables the device to enter recovery states that significantly improve observability compared to the existing scheme. We focus our experimental work only on Lazarus as it has an open-source implementation; therefore, our proposal's implementation is the extension of Lazarus on top of the same hardware and codebase.

To recapitulate, here are the contributions of our work :

1. First, we demonstrate the vulnerability of Lazarus to a message tampering attack on AWDT. We show that under a realistic assumption of an intermediate node being compromised, an adversary can render the device practically unusable by causing it to undergo a cyclic reboot.

2. Second, we show that Lazarus is vulnerable to software runtime attacks, which can similarly make the device unusable and create difficulties for the hub to take appropriate actions.

3. And last, to address these limitations, **we propose an updated design** based on the use of richer state-tracking. Our extension of Lazarus's implementation demonstrates improvement over the original scheme in terms of observability. This leads to better availability as the system can judge which attack happened at a particular time. Lastly, we provide performance figures and security analysis of our updated design.

In Section 2, we begin by presenting related work. The background information is provided in Section 3. Then, in Section 4, we describe the attacks we conducted on the original Lazarus implementation, which demonstrated the scheme's

susceptibility to network and software runtime attacks. Section 5 provides an analysis of the device state and its limitations in the original scheme.

In Section 6, we propose an improved design that utilizes the small TCB and introduces more robust state-tracking. We provide a detailed description of the new design and its implementation in Section 7, followed by an evaluation in Section 8. In Section 9, we discuss the security aspects of the new design, followed by a discussion about the real-world implementation of CIDER, Lazarus, and our solution in Section 10. Finally, we draw conclusions and anticipate future works in Section 11.

2 Related Work

The concept of recoverability is central to both CIDER and Lazarus. Our paper builds upon these principles, broadening the scope of potential attack circumstances in a quest to achieve better device availability. There is a body of work that also targets similar issues[Suz+20; AC20; Röc+21; Nun+20; IST19; De +22].

These studies typically aim to devise methods to remotely recover compromised devices, whether by guaranteed means or a best-effort approach. Our research enhances these existing mechanisms by introducing an additional layer of solution, resulting in a more resilient system for device healing.

Recent works have focused on improving the security and performance of IoT devices and networks at various levels. SA4P [De +24] proposes an IoT security framework that allows fine-grained control over IoT devices' interactions with the physical world through a Peripheral Guard (PEG) managed by a trusted deployment manager. TeeFilter [RBM24] is a formally verified network filtering engine for high-end IoT and edge devices that selectively executes parts of the network stack in a Trusted Execution Environment (TEE). PAISA [JKT23] ensures secure and periodic announcements of IoT device presence and capabilities to nearby user devices, leveraging ARM TrustZone TEE. ACFA [CRN23] is a runtime auditing and device healing system that uses active control flow attestation (CFA) to monitor control flow transfers and guarantee delivery of control flow reports, even if the device software is compromised.

A crucial component of the device healing mechanism is the capacity to attest to the software state of the device. In a dominance model, this attestation requires verification by an external entity. The concept of remote attestation was originally proposed by the Trusted Computing Group (TCG), which utilizes specialized TPM hardware. Currently, TCG has already standardized the next generation TPM, namely version 2.0, which was released in October 2014.

Recognizing the challenge of requiring specialized hardware in compact IoT devices, TCG has striven to meet these new requirements through the Device Identifier Composition Engine (DICE) standard [Gro21]. The DICE standard presents a robust architecture based on a combination of hardware and software.

It introduced a technique to generate attestation evidence during boot time, requiring a secret value obtained from the latched boot ROM.

The implementation of DICE has been showcased by [JPF17] and formally validated by [Tao+21]. Both the CIDER and Lazarus schemes discussed in this paper incorporate DICE as an essential element of their design. [LJF24] propose methods to accelerate DICE key generation on resource-constrained devices by caching keys or public key data, enabling faster boot times which is important in domains like automotive systems.

Remote attestation can be broadly categorized into two classifications: 1.) Software-based, and 2.) Hardware-based. The primary rationale for opting for pure software-based attestation typically relates to cost-effectiveness, given that no specialized hardware is required and the entire process is facilitated by code [SL19; Amm+19; Gri+22; ADD21; Sur+21]. These solutions often demand that the Microcontroller Unit (MCU) is equipped with a memory protection mechanism upon which the attestation service can be constructed.

Conversely, the hardware-based approach [Mae+18; ALK20; Vli+19] is typically associated with efficiency and security arguments. It enhances efficiency through the utilization of specialized hardware and bolsters security by executing on dedicated hardware designed exclusively for this purpose.

However, a hybrid approach combining hardware and software-based methods [Nun+19; ACT20; Hri+18] is now emerging, driven by the decreasing costs of hardware. The previously mentioned TCG DICE model is another example of this approach. This enables designers to optimize cost and efficiency to extract maximum benefits from both methods. The hybrid approach usually sets a minimum hardware requirement (which most MCUs are presumed to support) while simultaneously reducing cost by offering abundant features in the software.

There is also a new approach to transfer the responsibility of the sole trusted verifier to an untrusted intermediary, known as a proxy [Pet+22]. This proxy, fulfilled using a smart contract, assists in affirming provers' credibility on behalf of verifiers.

3 Background

This section walks through the main principles of CIDER/Lazarus. Both solutions are based on the DeferralTickets, `BootTickets`, and AWDT. The summary of CIDER and Lazarus that is relevant to our work can be seen in table 1.

3.1 CIDER

The idea of CIDER is to solve the problem of recovering IoT devices after a serious attack. The goal is to bring the devices back to a safe state unconditionally after such a fatal failure. This, in turn, provides the owner with the power to reclaim the ownership of the device even after the attack happens.

Aspect	CIDER	Lazarus
Open Source?	No	Yes
Hardware requirements.	Requires additional hardware features such as storage read-write latches and an external MCU for AWDT.	Aims to minimize hardware requirements by implementing latches, the reset trigger, and peripheral access control in software using a TEE.
Threat model in terms of communication between the hub and the device.	Both allow the adversary to eavesdrop on, tamper with, or block communications between devices and the hub. However, the adversary cannot indefinitely **block** the communication.	

Table 1: Comparison between CIDER and Lazarus

The solution is called *dominance* to reflect this power given to the device owner. Unlike attestation [GRB03; MWW02], which focuses only on confidentiality and privacy, CIDER is claimed to provide availability. This is manifested by the administrator's ability to specify a *guaranteed* firmware update in a scheduled time period.

CIDER requires both: 1.) storage latches to be able to write-protect recovery TCB from the untrusted world and 2.) AWDT implementation as a separate MCU. The former is implemented in the embedded multi-media card (eMMC) memory chips, which are somewhat complicated and costly. Apart from that, eMMC is naturally nonexistent on tiny MCU-based hardware. The latter results in a situation where the AWDT component has a similar complexity to the primary device it is planned to defend.

3.2 Lazarus

To overcome two issues on CIDER previously mentioned, Lazarus has been introduced with the same goal to unconditionally reclaim the device's ownership. The storage latches and AWDT feature are implemented in software running in a trusted execution environment (TEE). This way, it is possible to have AWDT without additional MCU and just rely on the existing Commercial-Off-The-Shelf (COTS) hardware, i.e., with a commonly used TEE, like ARM TZ.

As for the latch, Lazarus constructs the TEE to form the flash range that keeps Lazarus's code/data inaccessible from the untrusted world. Figure 1 depicts the boot flow of Lazarus. The green part is considered safe as it runs inside the TEE, but not the red area.

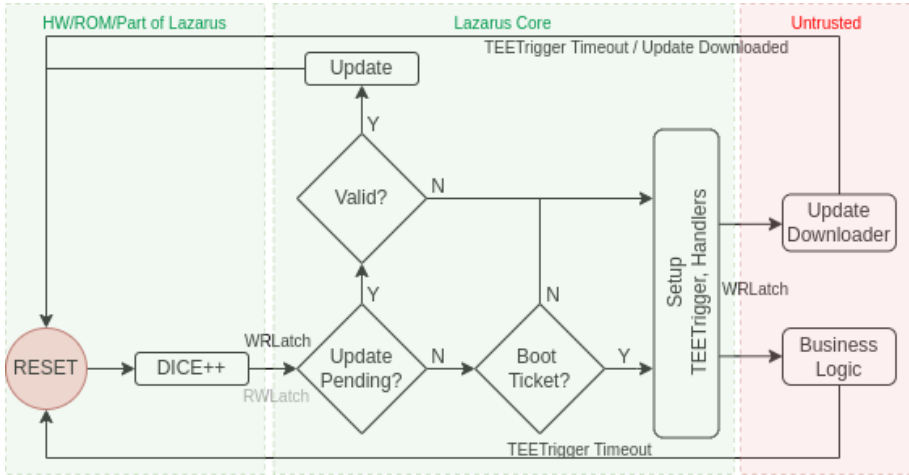

Figure 1: Boot flow of Lazarus.

If one disregards the details, the flow is similar to CIDER except the fact that Update Downloader is considered to be untrusted. The reason behind this design choice is that the downloader introduces a broad attack surface as it faces the internet.

4 Attacks Proof of Concept on Lazarus

In this section, we show two attacks against Lazarus implementation: (i) AWDT Ticket Attack and (ii) Runtime attack. Lazarus paper explicitly mentions the following attack vectors: (i.) *"Manipulate or block the communication channel between device and hub."*, (ii.) *"Attempt to tamper with Lazarus by interfering with its execution, by overwriting it or by forging updates."* within the scope of its design.

For the former, while the author narrowed down the scope of the DoS attack by claiming that the network attack could only last a maximum of 48 hours, an IoT device handling critical tasks (i.e., pacemaker, controller for a nuclear reactor, etc.) would not be able to tolerate such long time.

For the latter, since the device is completely dependent on the AWDT messages, a zero-day attack during runtime could affect the device's operation (if the zero-day affects the AWDT). In this section, we show that such a scenario could exist. The vulnerability itself is **not** Lazarus's vulnerability but comes from the underlying system that Lazarus uses.

We are showing the following shortcomings to **exemplify** that Lazarus's design cannot cope with such a scenario, and a better state handling could be achieved if the design is slightly changed. While the vulnerability can come and go, the

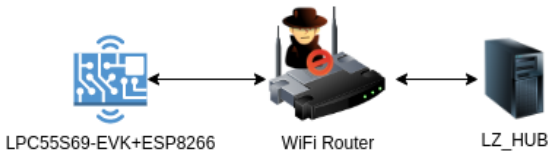

Figure 2: Attack against AWDT.

following proof of concept holds against the current open-sourced Lazarus implementation.

4.1 AWDT Ticket Attack

CIDER/Lazarus are two schemes claiming to provide strong protection against active attacks through robust recovery mechanisms. These mechanisms rely on the use of an AWDT, which is capable of forcing an IoT device to restart if its software is compromised.

However, the threat model of CIDER/Lazarus **explicitly** allows for eavesdropping, forging, and tampering (but not indefinitely blocking) with the communication channel between the hub and the IoT device. This is a significant concern in the context of IoT security, where malware is widespread on small home routers, making it easy for attackers to intercept or modify AWDT tickets through man-in-the-middle (MITM) attacks.

Consumer routers, cameras, and digital video recorders are among the most commonly infected devices by malware like Mirai [Ant+17b], with routers alone accounting for 17% of total infections. Given the prevalence of vulnerabilities in home routers and the ease of unauthorized access, it is not a stretch to imagine that middleboxes could potentially tamper with messages in transit, compromising the security of AWDT in CIDER and Lazarus.

In our experiment shown in Figure 2, we conducted an MITM attack on the Lazarus IoT device to test its robustness against active attacks. As there is no open-source implementation of CIDER, we utilized Lazarus's code [Fra22] (commit `7cf54702`) to conduct the attack.

We ran the source code on the NXP LPC55S69 [Sem19], and Espressif ESP8266 [Sys22] hardware, which was connected to the hub through a Raspberry Pi 4B acting as a wifi router. The router was able to eavesdrop and tamper with any passed-through message, allowing us to selectively choose when to tamper with the message.

During our attack, we tampered with the actual communication between the device and the hub, but not the TCP handshake, as the AWDT message is not sent from the device if the TCP is not established. We flipped the payload by only one bit, and as the payload was changed, the checksum needed to be recalculated.

The tampering rule was put into effect once the device had finished the booting process and started with the application execution. We allow the device to finish the booting process until valid software is verified to run inside without interceptions, after which the router initiates the process of the tampering rules.

Due to the router's tampering rules, when the device sent the deferral ticket, the hub could not verify its signature using the `AliasID` public key. As mentioned in Lazarus, the device is able to survive the reset if the next deferral ticket request is verified. However, as this was not the case, the device decided to reboot.

The device went through the execution of DICE++, followed by Lazarus core, and finally, the Demo App. When the Demo App was initiated, there existed a mechanism for waiting forever if `AliasID` update was unsuccessful. At this point, the device stopped working and did not perform any subsequent tasks.

However, as the watchdog timer period had already kicked in (the default value is 3600 seconds), the device underwent another reboot, followed by the same sequence of events. This hourly event occurred indefinitely, rendering the device practically unusable. The full logs of our attack on both the hub and device side are available in the repository [Ati23c].

4.2 Runtime Attack

In our analysis, we highlight an **existing** critical vulnerability in FreeRTOS v10.2.0, the operating system that Lazarus utilizes. Specifically, this vulnerability, documented as CVE-2021-31571[2], corresponds to an integer overflow within the queue allocation mechanism of FreeRTOS, which becomes exploitable when a queue is allocated within `sizeof(queue_t)` bytes of 4GB.

Given Lazarus's reliance on FreeRTOS v10.2.0 at the time of its implementation, it inherently inherits this vulnerability. It is crucial to note that the vulnerability itself originates **not** from Lazarus but from the FreeRTOS version it employs. This exemplification underscores that should such a scenario occur, Lazarus would be susceptible to the implications of this pre-existing vulnerability, especially if the vulnerability is of a type of zero-day.

To further illustrate the vulnerability's implications, we modified the Lazarus framework to allow dynamic queue sizing for its network interface. In typical scenarios, a remote IoT device may experience fluctuating connectivity. Under stable connections, outgoing packets can be directly transmitted without queuing. In contrast, during unstable connections, increasing the network interface's queue can help manage packet traffic.

The default Lazarus implementation has a fixed queue size of 8096. Our modification involved altering the function lzport_usart_init_esp() to accept an integer input, which subsequently is passed to lzport_esp8266_init_queue() and eventually consumed by xQueueCreate(). It is important to highlight that this function is

[2]https://nvd.nist.gov/vuln/detail/CVE-2021-31571

where the buffer overflow vulnerability resides. Also, any change to the interface's queue size mandates a device reboot.

In an extreme connectivity condition, a device operating with the default queue size of 8096 needs to adjust this value. If, for instance, the device alters its queue size to 1 GB (1073741824) in response to intermittent connectivity, it would encounter the buffer overflow vulnerability when trying to initialize the interface. This results in `xQueueCreate()` returning `NULL`, throwing the `ERROR: Failed to initialize ESP queue`, rendering the device inoperative until the subsequent reset, which is set to a default of one hour.

Should the queue size remain unchanged post-reset, the device would enter a continuous reset cycle, reminiscent of the previous attack described in section 4.1. Since no network interface is available, the hub cannot serve the AWDT. Lazarus's change to achieve our purpose in showing CVE-2021-31571 can be seen on the following repository [Ati23d].

Step	Description
1	Modify the Lazarus framework to allow dynamic queue sizing for its network interface.
2	Alter the function `lzport_usart_init_esp()` to accept an integer input, which is passed to `lzport_esp8266_init_queue()` and eventually consumed by `xQueueCreate()`.
3	Under extreme connectivity conditions, change the queue size to a large value, such as 1 GB (1073741824).
4	When the device attempts to initialize the interface with the modified queue size, it encounters the buffer overflow vulnerability in `xQueueCreate()`, causing it to return `NULL`.
5	The device throws the error "ERROR: Failed to initialize ESP queue" and becomes inoperative until the next reset, which is set to a default of one hour.
6	If the queue size remains unchanged post-reset, the device enters a continuous reset cycle, rendering it unusable.

Table 2: Steps of the runtime attack exploiting CVE-2021-31571.

To summarize the steps of the runtime attack exploiting CVE-2021-31571, consider table 2. Notably, the integer overflow vulnerability can be exploited (or accidentally triggered) by anyone who has access to the Demo App, which is a userspace application residing in unsecured storage. Knowing the existence of a vulnerability in the underlying system for device recovery is crucial, especially if it can affect the runtime of the system and result in a possible runtime attack, such as CVE-2021-31571.

Even though the vulnerability may come and go, it is still essential to be aware

of its presence, as it could lead to repeated platform resets and render the device unusable. The potential consequences of such attacks can be severe, and it is essential to take appropriate measures to mitigate them.

5 CIDER/Lazarus Main Shortcomings

In this study, we conduct an analysis of the security and availability aspects of the CIDER/Lazarus designs, with a particular focus on their potential improvements and identified limitations.

5.1 Dominance Analysis and Potential Improvements

The fundamental concept underlying the dominance scheme involves granting a central hub the ability to exert control over a device by issuing regular tickets to it. As long as the device continues to receive these tickets, it operates normally. However, if the ticket supply is disrupted, a watchdog function takes over and initiates a device reset. During the reset process, the TCB executes the boot code to ensure that the hub can regain control, perform updates, and carry out other necessary functions.

Nevertheless, our analysis, as presented in Section 4, demonstrates that the ticket-serving mechanism, which is crucial for maintaining hub dominance, is susceptible to two distinct types of attacks: (i) A direct, network-based attack on the tickets, which will force a reset through the AWDT mechanism as we showed in our attack example in Section 4.1. And (ii) A software run-time attack that will influence the ticket serving mechanism that can utilize a potential weakness in the TCB part of the implementation, as we showed with our example in Section 4.2.

The first type of attack renders the device completely nonfunctional as long as the attacker persists in interfering with the ticket-handling process. However, the second type of attack offers the possibility of mitigating the impact. By switching to an unaffected code base and detecting the ongoing runtime attack, the system can take appropriate measures, such as performing a firmware update.

Consequently, maintaining control and dominance over the hub becomes feasible in the second attack scenario. Achieving this requires implementing richer device state tracking and separating different code components within the TCB. By adding a separate mechanism from the existing one, the design ensures that the device's functionality is not solely dependent on the integrity of the entire TCB.

This approach offers a twofold advantage: (i) The new design ensures the maintenance of hub control even in the face of severe runtime attacks, including zero-day exploits. And (ii) Furthermore, the design enables the identification of a device that is likely to be under a network ticket attack. Once detected, appropriate actions can be taken to mitigate the impact of the attack.

If possible, the device can switch to an alternative network connection, evading the ongoing attack and maintaining its operational status. Alternatively, if

switching the network connection is not feasible, the device can be transitioned into a minimally functional state. This state allows the device to remain operational during the attack, albeit with limited functionality, ensuring the continued operation of the device.

To demonstrate the effectiveness of the proposed design, we present its details in Section 6. Additionally, a proof of concept implementation is provided in Section 7.

5.2 Boot Tickets Usage Analysis

The CIDER and Lazarus schemes incorporate a novel concept known as "`BootTickets`" to streamline the boot process. By leveraging `BootTickets`, these schemes enable direct booting of the firmware without the requirement of initial attestation to an external trusted hub following a platform reset.

This mechanism operates on the basis of a `nonce` that is securely stored during gated boot and subsequently included in a DICE-attested request sent to the hub. In response, the hub issues a boot ticket containing the nonce and an acceptable firmware image hash value. The primary objective behind adopting `BootTickets` is to attain platform state verification in subsequent boots without the need for repetitive attestations, resulting in significant time savings.

The principle of dominance stipulates that the hub maintains comprehensive oversight of the devices and their associated software state. Thus, the hub consistently has complete knowledge of each device's anticipated software state. As long as there exist no network disruptions, the hub may exercise control over the device via the `DeferralTicket`. It's straightforward to augment this ticket with information determining whether the hub necessitates a reboot with attestation from the device. For Lazarus, this information is subsequently processed by the TCB, and for CIDER, the AWDT hardware performs the task.

The local verification against the latest accepted firmware image is better done with a traditional secure boot certificate [Pro+19] than using boot tickets. Such a certificate is then distributed together with the image. A new certificate will then be included at each firmware upgrade, and the hub will be sure which latest expected firmware image hash is used for local verification at each reboot. This is then the very same hash value that is included in a boot ticket.

Furthermore, the usage of `BootTickets` will require *two* consecutive reboots prior to a new attestation when the hub decides that such will be needed, which is less efficient than directly requesting it through an information flag in a `DeferralTicket`. If the `DeferralTicket` is not received in time, there is indeed a severe problem with the dominance, and reconnecting with the hub is the main security priority. The hub should then decide whether a new attestation is needed or not.

In summary, the combination of more extensive attestation control in the `DeferralTicket` and a secure boot verification will give the same effect as us-

ing a `BootTickets`. This reality, together with the fact that `BootTickets` can also easily be modified using network attacks (as we showed in section 4.1), are arguments for removing the `BootTickets` altogether in the design and instead let the hub directly control the reset behavior through the `DeferralTicket` *only* as we do in our updated design.

6 The New Design

CIDER/Lazarus has established a foundation for a device recovery mechanism. The principle of *dominance*, where a trusted hub has the main power of the device through utilizing secure hardware or the TCB of the device, is a powerful concept that allows the hub to take control whenever the device does not operate normally. This is indeed true, regardless of whether the reason for a service disruption is an error or an attack.

However, as we have shown in Section 4 and 5, the ticket usage itself is very sensitive to network and runtime attacks, which causes the hub control to fail. Furthermore, if the hub loses contact with the device, it will not know if the reason is a problem with the ticket delivery or a software compromise or error. Hence, to actually live up to the promise of hub dominance, there is a need for a more robust design with better control possibilities from the hub as well as richer device state tracking.

To enhance the robustness of CIDER and Lazarus, we present an improved design that addresses these issues, allowing considerably improved device dominance, as shown in Figure 3. We propose a validation mechanism to determine whether the inability to retrieve AWDT is attributable to external factors, such as network disturbances, or internal, like runtime attacks. We do this by introducing richer state tracking.

Especially when a ticket verification or fetch fails, we keep track of if this occurred during the "normal" operation, and then after reset, instead of continuing to normal operation, we move into a *new* mode, LZ_VERIFY, with protected and separated code/execution. If valid ticket fetch also fails in this mode, we can, with a high likelihood, conclude that the main problem is a network problem and not a runtime attack, and we can act accordingly (repeated tries and or moving to a minimal operation mode).

However, if it succeeds, a runtime is likely, and we can take proper actions, inform the hub about the conditions, and ask the hub to take proper action (like requesting a software update or similar). The primary goal of the new design is to ensure the highest possible level of availability, even under attack.

As discussed above, the verification process is executed through a new `boot_mode`, called LZ_VERIFY. This serves as a component in enhancing observability when the application (APP) detects a problem (internal or external). An internal problem implies that the APP has been compromised and requires an update, irrespective of the hub's readiness.

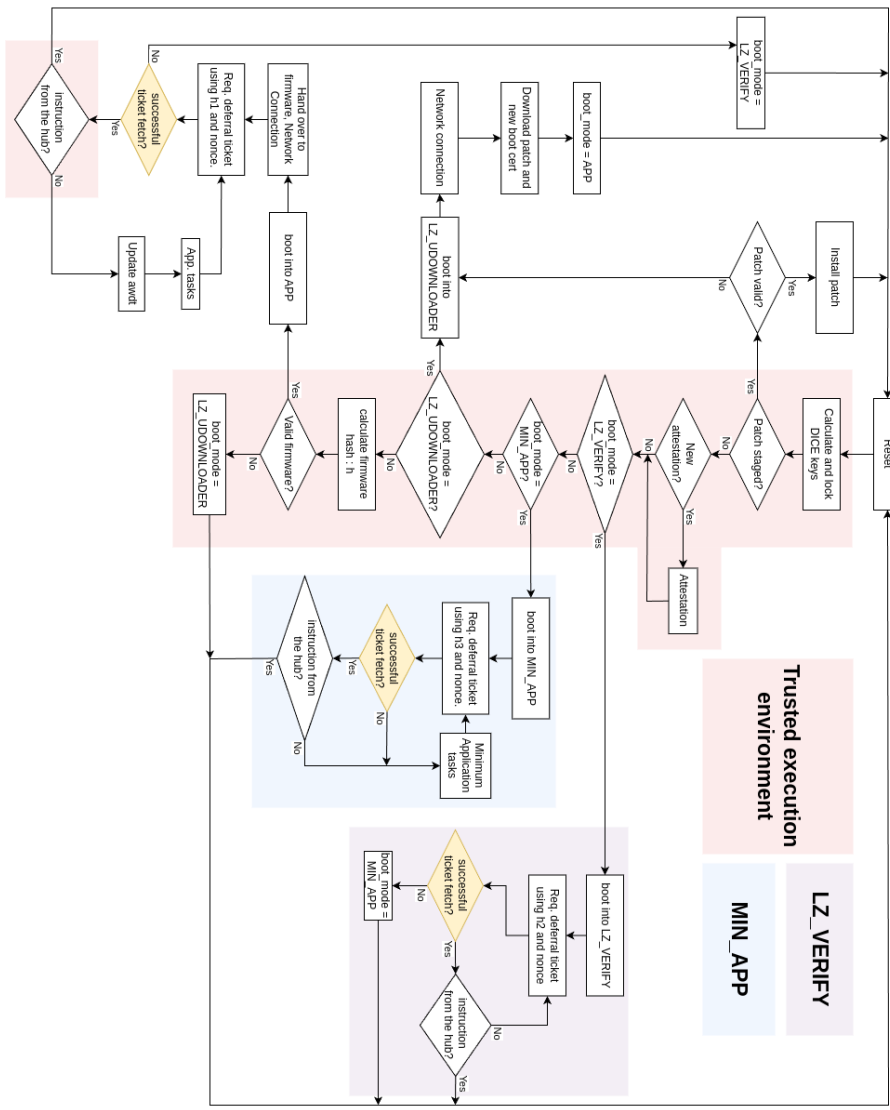

Figure 3: The New Design.

Conversely, an external problem suggests that the device must relinquish its dependency on the AWDT, provided that the occurrence of the external problem is verified. This serves **ONLY** as the final contingency measure when both the device and the hub are at a loss for further actions.

To recapitulate, in Lazarus, various states denoted as `boot_mode` are utilized. These states include:

- **APP**: This state encompasses the portion of the device that houses the business logic.
- **LZ_UDOWNLOADER**: The device boots into this to establish communication with the hub and do firmware updates.
- **LZ_CPATCHER**: A core patcher responsible for applying Lazarus Core updates. As this mode is unrelated to the issue discussed in this paper, we exclude it from our analysis. For a detailed explanation of LZ_CPATCHER, readers are referred to read the Lazarus paper.

In addition to the existing `boot_mode` states, we introduce the following new states to accommodate our proposed solution:

- **LZ_VERIFY**: This mode is intended to be invoked in the event of a disruption to the device's operation, assuming the hub has not scheduled an update. As previously discussed, the cause of the disruption may be internal, such as a runtime attack (which the hub and device are typically capable of resolving collectively) or external due to the device's reliance on the AWDT, which requires constant network connectivity.

  Given these, the security of the LZ_VERIFY codebase must be of paramount importance. One plausible approach is to locate this code within the TEE section of the device. While this placement is feasible in the Lazarus model, it may not be in the CIDER model. However, if device limitations make this impractical, LZ_VERIFY could be positioned within an unsecured section, provided it remains unaltered at all times. Additionally, the fact that LZ_VERIFY can only be executed following a platform reset ensures the continuous ability to verify the software's hash prior to its execution.

- **MIN_APP**: When the device cannot retrieve DeferralTicket from the LZ_VERIFY (making it inaccessible from both APP and LZ_VERIFY), the device concludes that there exists a network attack. By this far, there is no point in trying to operate normally until the network is back. In this state, the device expends minimal computational effort to maintain usability while the attack is ongoing, hence called MIN_APP.

  The definition of "minimum" may vary depending on the specific application. Determining which computations to prioritize is beyond the scope of

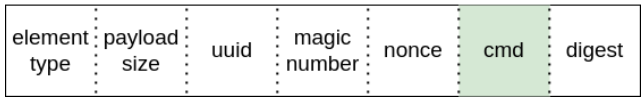

Figure 4: Additional field (green) in the signed area of the message header.

this paper, and we defer to application developers for such decisions. However, it must preserve the core computation resource of the device according to its purpose without the need to contact the hub through all the AWDT processes. Periodically, the device checks for the restoration of a connection to the hub. If a connection is reestablished, the device can return to the APP state (subject to the hub's authorization).

Figure 3 shows the interaction between components in our new proposed design, where the purple and blue sections are the LZ_VERIFY and MIN_APP, respectively.

6.1 Best Effort Dominance

This paper focused on maximizing the usability of IoT devices by granting them a more active role in determining when and how to respond to external circumstances, such as compromised firmware or network attacks. Unlike CIDER/Lazarus, where the hub solely acted as an integrity-acknowledgment service for device messages, our solution empowers the device itself to make decisions when the hub is inaccessible.

However, when the hub is reachable, we anticipate the device will be in complete compliance with the hub, maintaining the principle of full hub dominance. While CIDER/Lazarus assumes a binary view of surrounding circumstances, limited to normal situations and continuous network attacks, we recognize that adversaries may exhibit more complex behavior, requiring decision-making beyond the input received by the device alone.

To address these challenges, we introduce a concept called "*best effort dominance*". By leveraging the hub as a centralized entity, we aim to provide better control and influence. Instead of an **unconditional** dominance, our approach offers the **best effort**, wherein the hub can send commands to the device as long as the message integrity is maintained. By embedding information on which state the device currently is into the `DeferralTicket` request, the device expects to receive instruction on what to do next at any given time.

A scenario where best-effort dominance is vital involves changes in application behavior (APP) due to changes from adversaries. As previously mentioned, an adversary can modify the application's behavior while keeping the `DeferralTicket` intact, allowing the device to operate incorrectly without detection.

Since the hub is the only entity capable of detecting such deviations, it can initiate commands to reset the device. If the abnormal behavior persists, the hub

can command the device to perform a reset with attestation. In the Lazarus design, this can only be executed during the next platform reset, initiated by the device when the AWDT expires, while the hub passively remains inactive.

The same mechanism can also serve as a substitute for `BootTickets`, expediting the boot process due to the hub's capability to instruct the device to reset with or without an attestation at any given moment. Even within the LZ_VERIFY and MIN_APP modes, as long as the device can receive a verified instruction from the hub, the hub retains the ability to guide the device's subsequent actions to the best of its capacity.

Conversely, if the integrity is compromised, it is impractical to insist on dominance, i.e., if the adversary intermittently tampers with the message, continuous communication with the hub may be futile. In such cases, it is preferable for the device to follow the hub's instructions independently.

To enable this, the hub can command the device to remain in a specific state, such as the MIN_APP, for the duration of the network attack. The hub can employ various mechanisms, such as network probes, to detect the occurrence of network attacks and decide on appropriate actions for the device. In the most extreme scenario, even if the message instructing the device is forged, the device will still transition to the MIN_APP mode, as illustrated in Figure 3.

To facilitate these functionalities, we introduce an additional field in the header of response messages from the hub. This command is not initiated by the device itself but rather piggybacked in a reply message, i.e., in response to the `DeferralTicket`.

Figure 4 illustrates the modified message header, which includes a field for controlling the device. In situations where the message integrity is compromised, the device is not obligated to comply with the commands from the hub. However, the device's obedience to these instructions is essential in normal circumstances.

6.2 Multiple Access if Possible

If possible, supplementary measures can be introduced to tackle the retrieval of a valid deferral ticket by adding interfaces and/or hubs if the primary attempt is unsuccessful. This is to confirm that a failed retrieval does not definitively signify a network attack.

The causes of unsuccessful ticket retrieval could range from a malfunctioning interface connection to a deactivated hub. By using multiple interfaces and hubs, system designers can attain a trade-off between cost and availability, optimizing their system in accordance with these factors. This strategy can be implemented in each yellow branching point from Figure 3.

6.3 Maintaining Autonomy when Dominance is Compromised

In the event that the dominance mechanism is entirely disrupted due to network disturbances, the device retains the capability to preserve its functionality.

Figure 5: The flash memory new layout.

It achieves this by independently transitioning into the MIN_APP mode from LZ_VERIFY. This action is reserved for scenarios where all prior attempts to maintain dominance have been exhausted.

7 Implementation

To demonstrate the feasibility of our proposed design, we have developed a proof-of-concept. To enable a direct comparison with the previous design, we have modified the Lazarus implementation and utilized the same hardware platform. The codebase of the improved solution is available at [Ati23b].

Specifically, we selected the LPCXpresso55S69 development board from NXP, which comes equipped with the LPC55S69 MCU, featuring a dual-core ARM Cortex-M33 with TrustZone-M support. To establish network connectivity with the hub, we employed additional hardware, the ESP8266, as the main board lacks native external network connectivity. The ESP8266 module is interfaced with the LPCXpresso board through its universal synchronous and asynchronous receiver transmitter (USART).

Our prototype carries the original allocation of the secured flash memory. Hence, the size of DICE++, Lazarus Core, and Core Patcher has not changed in any way. For the unsecured application, we shifted the flash so that the new `boot_modes` can be accommodated. The layout of the new design is depicted in Figure 5.

MIN_APP. The implementation of the minimum application is similar to the business logic application but without the need to refresh the AWDT. There is a mechanism to check the connection towards the hub occasionally. However, the purpose of the execution is not to get the AWDT working with Lazarus Core for a long period of time. But rather for testing the connection and going back to the normal state if it is deemed that there is no disturbance in the network.

In our implementation, the definition of minimum is that the device is able to use the sensor and save measurements to local storage. There might be a case when this is not enough, but we let the developers define what they need, depending on the application requirement.

LZ_VERIFY. The verification mechanism in *LZ_VERIFY* is an important component of the proposed design. Like the update downloader section in

Lazarus, the Flash Memory Controller (FMC) handler ensures that LZ_VERIFY can only write to specific areas in the flash memory. In our current proof of concept, we prioritize testing the principle on the very same platform as Lazarus to make fair comparisons between the two approaches.

However, as the secure memory in Lazarus was not enough, we were forced to implement LZ_VERIFY in the unsecure part. This allows us to test and verify the attack behavior of our new solution compared to the old, but in a real deployment, LZ_VERIFY must be implemented in the secure part of the flash together with the Lazarus Core.

To be able to change the boot_mode, MIN_APP and LZ_VERIFY can set a request on what to boot the next time the device decides to restart. This way, the LZ_CORE will receive the signal on what to boot depending on the request from the unsecured application.

8 Evaluation

The robustness improvement in this paper is based on the condition of network attack or runtime attack. Having said that, we evaluate our solution in the following scenarios: 1.) The network is compromised but not the firmware; 2.) The firmware is compromised, but not the network. Readers who are interested in learning the baseline of Lazarus' performance should read the original paper.

We follow a similar hardware setup used by Lazarus, as mentioned in section 7. Additionally, the hub ran on an HP EliteDesk PC, equipped with an Intel Core i7-6700 processor and 16 GB of DDR4 RAM, which connected to the wireless router through a 1Gbps network connection. The PC is installed with Ubuntu 18.04 and supported with Python 3.8.0. A Raspberry Pi 4B acts as a wireless router, which is also connected to the hub through a 1Gbps ethernet connection.

8.1 Device Behavior Under Network Attack

In this section, we discuss and evaluate our new design proposal only on a specific circumstance, i.e., a network attack.

Measurement on Vanilla Implementation of Lazarus

First, we measure the impact of a tampering attack on the original Lazarus. The impact is measured by how long the device can be in an operational state (APP) before becoming unusable based on the attack we mentioned in section 4.1. We ran the code without changing anything, meaning all variables are on the **default** value. Lazarus invokes three important variables during this attack:

$x1$ Default WDT timeout: 3600 seconds.

$x2$ Deferral ticket time: 60 seconds.

$x3$ Deferral ticket task wait time: 30 seconds.

It means that prior to any interaction with the hub, the device is able to survive based on $x1$'s value before the Lazarus Core forces the device to reboot if no `DeferralTicket` request is granted. Once the `DeferralTicket` is requested, $x2$ is invoked as the requested time, and if the hub grants the request, the device is waiting for $x3$ period before it sends another `DeferralTicket`, and so on.

Looking back at figure 1, the adversary starts tampering with the message once the device is fully booted, i.e., reached the business logic (APP) with one or more `DeferralTicket` is allowed to pass through the router without getting tampered. At this point, the WDT timeout has changed to $x2$ while the device continues to send the `DeferralTicket` request every $x3$.

After $x3$, the device sends a `DeferralTicket` request, but the hub refuses to authenticate the ticket as its integrity is compromised. The device is able to survive one more WDT timeout period ($x2$) before rebooting itself. As it reboots, the WDT timeout is back to the default value ($x1$). But, since the tampering rules persist, the integrity of all the messages is compromised, resulting in the hub refusing to update its AliasID certificate and repeating the cyclic reboot every $x1$. At this stage, the device is practically unusable.

Given that all the variables are unchanged, it takes 59.83 seconds for the device to go into an unusable state when a persistent message tampering attack happens. To make a fair comparison with our solution, we measure the operational state using the same time-bound, utilizing the value of $x1$. In 3600 seconds, the device is in an operational state for 59.83 seconds or 1.66%. At this point, even though $x1$ is changed to, i.e., shorter time, it does not matter, as it will just change the cyclic period of the device reboot, and the device is still unusable.

Measurement on the New Design

Next, as we know how long it takes to put the device into an unusable state, we compare them with how the new design would react regarding the "operation time". It should be noted that MIN_APP is **not** a fully operational state, but compared to CIDER/Lazarus, it is a state where the device can preserve its minimal function and critical parts, while CIDER/Lazarus is unusable and is not able to preserve anything under the same attack. One can see MIN_APP as similar to APP, only without the dependency on the AWDT message.

To get a fair comparison with the vanilla implementation, the transition period when the device shifts from APP to MIN_APP is not operational time. So, operational time is equal to APP+MIN_APP, while LZ_VERIFY is not included. This measurement uses the same default value as the previous one. When the same attack happens, the device can survive for $x2$ seconds due to the same reason.

However, as described in section 6, it verifies the occurrence of the network attack through LZ_VERIFY. If that is the case, then the device goes to the

	Lazarus	The New Proposal
Operational Time	59.83 seconds	3598.31 seconds*
Percentages	1.66%	99.95%

* Assuming that MIN_APP is counted as an operational state.

Table 3: The operational time of Lazarus and the new design in a one-hour timeframe.

Runtime Attack Target	Lazarus	The New Proposal
Application	No	Yes
Ticket	No	Yes

Table 4: Device Recoverability During Runtime Attack.

MIN_APP and stays there as long as the attack persists. The transition time between APP-LZ_VERIFY-MIN_APP took 1.69 seconds. In 3600 seconds timebound, the device is in an operational state for 3598.31 seconds or 99.95% of the total time. Table 3 sums up the comparison of the operational time between Lazarus and our newly proposed solution.

8.2 Device Behavior Under Runtime Attack

This section evaluates our new design during a runtime attack. The attack is either targeting: 1.) the application, or 2.) the `DeferralTicket`. As explained in 5, an unnoticed vulnerability (i.e., zero-day) can be exploited by an adversary to change the behavior of the device. Depending on the attack target, the outcome behavior can be different.

As shown in table 4, a runtime attack against the application or the `DeferralTicket` cannot be recovered by Lazarus. On the other hand, the new proposal is able to recover the device by utilizing the new field in the response message from the hub, called the "command".

8.3 Overhead of the New Signaling Message

To get a best-effort dominance, the hub must be able to send instructions, commanding the device on what to do, as explained in section 6.1. An additional command field is needed to serve this purpose, and the size can be adjusted based on the number of instructions. For the purpose of tackling the attacks we mentioned, 4 bytes is enough to cover all the instructions, as shown in table 5.

8.4 Complexity Analysis

The new design is more complex than the old Lazarus design as it covers a richer state to overcome the attack. It incorporates 23209 additional lines of code com-

	Lazarus	The New Proposal
AWDT Response Size	180 bytes	184 bytes

Table 5: AWDT response size comparison between Lazarus and the new proposal.

pared to the original Lazarus codebase. It is important to note that most of these new features are only invoked when the device is under attack, minimizing the impact on overall system performance while saving the device from cyclic reboot at the same time.

Under normal circumstances, LZ_VERIFY and MIN_APP are not invoked, resulting in no additional complexity to the system operation. However, when (i) the network is under attack or (ii) the firmware is compromised, these two functions may be triggered. In the case of the former, it depends on the attack frequency, while the latter depends on whether the compromised firmware affects the AWDT functionality or not.

LZ_VERIFY and MIN_APP are not significantly more complex than the original deferral ticket request in the Lazarus design. LZ_VERIFY's additional complexity lies in its ability to receive a 4-byte instruction from the hub during operation. MIN_APP is expected to be less complex than the original APP, as unnecessary functionalities are removed when the device preserves its resources. Overall, the added complexity is minimal, and most of these components are only invoked during an attack.

9 Security Discussion

Next, we analyze the security properties of the enhanced Lazarus design that we have provided. In our analysis, we follow the attack-oriented reasoning used by the authors of the Lazarus paper. For convenience, we here repeat the listing of the main considered attack vectors from [Hub+20]:

A-1 *"Manipulate or block the communication channel between device and hub."*.

A-2 *"Attempt to tamper with Lazarus by interfering with its execution, by overwriting it or by forging updates."*

A-3 *"Tamper with peripherals to permanently render devices irrecoverable. Examples are wear out of flash storage, shutting devices down or setting them into irrecoverable low-power states, or manipulating the firmware of other peripherals."*

A-4 *"Prevent or defer device reset, e.g., by manipulating or turning off TEETrigger or by forging deferral tickets."*

A-5 *"Inject malware into untrusted software and try to persist it across resets."*

A-6 *"Deceive the hub about the application of updates to Lazarus."*

Below, we discuss how or to what extent our new design mitigates these main threats.

Attack type A-1

As explained in section 4.1, Lazarus cannot defend an attack against forged, tampered, or blocked communication between the hub and the device, particularly when such malicious actions affect AWDT messages. The discussion on how we solve this (to the best possible extent) can be seen in section 9.1.

Attack type A-2

As explained in section 4.2, Lazarus cannot distinguish (and also safeguard) against runtime attacks targeting the application and/or the AWDT. We elaborate on our solution to this problem in section 9.2.

A-3

Our design inherits Lazarus' approach of using trusted handlers to control access to critical peripherals. By ensuring that the application interacts with peripherals through these secure handlers, we maintain the same level of protection against tampering attempts.

A-4

The use of a hardware-based secure element (TEE) to trigger device resets, as employed in Lazarus, is preserved in our design. This makes it difficult for attackers to manipulate or disable the reset mechanism, ensuring timely recovery in case of compromised functionality.

A-5

Our design follows Lazarus' approach of storing critical components like DICE++, Lazarus Core, and Core Patcher in secure memory regions. This isolation, combined with the attestation process during boot, helps prevent malware from persisting across device resets.

A-6

We maintain Lazarus' secure communication channels and attestation mechanisms between the device and the hub. This ensures that the hub can reliably verify the integrity of the device's software state and detect any deception attempts regarding updates.

The new design follows the Lazarus and CIDER design with the following exceptions:

- Richer state tracking
- Introduction of a new mode: LZ_VERIFY
- Modified ticket handing by replacing boot tickets with `DeferralTicket` only

The richer state tracking is achieved by tracking the execution state in the *trusted* and protected part of the system. This is like the design in Lazarus, and there are no additional threat vectors with respect to properties A-3 until A-6. The code has been updated to make sure no deadlock conditions arise.

The new mode, LZ_VERIFY, is a separate mode but, apart from that, contains the same ticket-handling code as before. Consequently, the previous security analysis applies also to LZ_VERIFY.

Regarding the new control commands in the ticket, those are parsed by the trusted part of the code. The security of the control commands is in the hands of the trusted code base. Special care needs to be taken to make sure this code fulfills the security requirements in order not to influence the overall security.

The next part of this security discussion will be about solving the two shortcomings in the previous design with our extended solution to resolve both claims in A-1 and A-2.

9.1 Building Awareness when Network Disruptions Occurred

The critical issue underlying the cyclic reboot and practical unusability of the device in CIDER/Lazarus, as discussed in Section 4.1, stems from the inability to detect network disruptions. This limitation arises due to the complete reliance on AWDT for normal device operation **at all times**.

We have emphasized that this combination of premises proves problematic when both are simultaneously encountered. While the concept of AWDT is sound, during network disruptions, it is more effective to relinquish dependency on AWDT and strive for normal device operation without it.

To address this challenge, we introduced MIN_APP, enabling the device to conserve resources and maintain limited functionality during network attacks. However, the decision to transition into MIN_APP should be made carefully, without yielding too easily to entering the minimum state whenever convenient. Instead, the occurrence of a network attack must be confirmed through an alternative `boot_mode` outside of APP, namely LZ_VERIFY, which is presumed to offer greater security than APP.

Only after the attack is verified can the device transition into MIN_APP mode. The role of LZ_VERIFY is to verify whether the event constitutes a network attack or another anomaly originating from within the device. An internal anomaly does

not warrant the activation of MIN_APP; instead, a software update (if available) should suffice to address such anomalies, as mentioned in 9.2.

In addition, ensuring the appropriate decision-making process for transitioning between MIN_APP and APP modes is crucial to reflect the conditions accurately. A clever adversary could strategically manipulate message tampering intermittently to deceive the device into believing that network disruptions have ceased, prompting it to switch back to APP mode even when disruptions persist. This can lead to subsequent troubles triggered by the adversary. To address this challenge, we propose the following:

1. MIN_APP continuously monitors the network connection status at the end of each period. Let t represent the duration of each period, where $t = 2^r$, and r is initially set to 0.

2. If, at the end of period t, the connection to retrieve AWDT is restored, r is incremented by 1, thereby increasing the duration of the next period exponentially whenever the device attempts to transition from MIN_APP back to APP.

3. The successful retrieval of AWDT must be confirmed through a second opinion from the hub. This confirmation is achieved by utilizing the `cmd` field, as mentioned in figure 4, where the hub sends a message to validate the reestablishment of the connection.

4. Only when both conditions are met (i.e., the connection is restored and confirmed by the hub) can the device transition back to normal APP mode. Otherwise, it must remain in MIN_APP for an extended period based on the exponential duration determined in the second point.

By implementing the proposed mechanisms, we significantly enhance the lifecycle operation of the device, specifically addressing the problem when the device is faced with an attack targeting AWDT and, ultimately, live up to the expectation of attack-vector A-1. Regardless of whether the network disruptions occur intermittently or continuously, we effectively minimize the impact on device operations, in contrast to previous solutions offered by CIDER/Lazarus, which rendered the device practically unusable.

9.2 Giving the Device a Better Observability to Differentiate Runtime Attacks

In regards to A-2, we have identified three distinct impacts that a runtime attack can have on the device, depending on its target:

T-1 Impact on AWDT: The attack aims to disrupt the functionality or integrity of the AWDT. This type of attack directly affects the device's recovery mechanisms and can undermine its ability to maintain normal operations.

T-2 Impact on the device behavior (**known** by the hub): The attack specifically targets the device's behavior from the userspace or application perspective. In this case, the hub is assumed to know the attack and its potential consequences. The information that enhances the hub's awareness is conveyed via the `DeferralTicket` request, in conjunction with changes in the APP behavior that can be detected from the backend application at the hub. This type of attack may alter the device's intended functionality or introduce malicious behavior that is recognizable by the hub.

T-3 Impact on AWDT or device behavior (**unknown** to the hub): The attack affects either the AWDT **or** the device's behavior, but the hub is unaware of the attack and lacks knowledge on how to mitigate its effects at the moment of attack. This type of attack poses a significant challenge as it compromises the recovery mechanisms and can introduce undesirable behaviors in the device without detection or effective countermeasures from the hub.

In the case of T-1, where the AWDT is affected by a runtime attack, the device becomes unable to fetch the AWDT from the APP. In our extended design, this event triggers the device to transition to LZ_VERIFY. Note that the AWDT is still fetchable from the LZ_VERIFY state, as it remains unaffected by the runtime attack.

This ensures the device can continue its recovery procedures and maintain its integrity. Following the LZ_VERIFY state, there are two potential options for the device depending on the situation:

- Hub aware of the situation (T-2): As the hub has the awareness, the software update to solve the problem is presumably available. The hub asks the device to transition to the LZ_UDOWNLOADER state. In this state, the device retrieves the necessary software update and proceeds with the update process, ensuring it remains up-to-date with the latest security measures and fixes.

- Hub unaware of the situation (T-3): In scenarios where the hub is unaware of the impact of the runtime attack, the hub may, in the worst-case scenario, instruct the device to transition to the MIN_APP state until the issue is resolved. The `cmd` field, as discussed earlier, can be utilized to relay instructions from the hub to the device. By instructing the device to stay in the MIN_APP state, the hub can prevent further potential risks or consequences resulting from the unknown behavior shift.

The impact of T-1 on device AWDT fetching is notable, whereas T-2 does not display a similar influence. In the event that a software update is scheduled, it will be automatically initiated during the subsequent AWDT fetching period through the instruction from the hub, provided that the update is available. The hub may

or may not ask the device to transition to the MIN_APP, but such a decision is contingent upon the severity of the impact.

In contrast to the approach taken by CIDER/Lazarus, where both runtime attacks targeting AWDT and communication attacks affecting AWDT result in the device becoming unusable due to their heavy reliance on the AWDT message, our solution introduces improved action points for the device based on its awareness of the specific circumstances at any given time. This approach ensures that the device does not remain stagnant or completely non-operational, even when facing different types of attacks.

In the worst scenario, when the hub is unaware of the issue, our solution allows the hub to ask the device to preserve its resources while still maintaining core functionality by transitioning to the MIN_APP. This request from the hub is triggered only under a limited time, i.e., until i) the hub identifies the problem and ii) a software update is available.

On top of that, the capability to transmit instructions from the hub to the device enables the system to respond appropriately at the earliest opportunity (i.e., subsequent to each `DeferralTicket` response). Since all the `boot_mode` scenarios are equipped with this functionality, the hub can continually send instructions, provided the integrity of the message is preserved, paving the way for better dominance, but in a best-effort manner.

By incorporating these, our solution ensures that the device remains operational and adaptable to various attack scenarios. It strikes a balance between resource preservation and maintaining operational capabilities with the best-effort dominance, allowing the device to continue functioning properly even in the face of adversarial actions.

This way, the claim in attack vector A-2 can live up to the expectation as the device has better observability and can act accordingly based on a more informed decision.

10 Real World Implementation

Similar to the argument from the Lazarus paper, practical integration of CIDER, Lazarus, and our extended solution requires the developer to configure the solution according to the device's peripherals and the specific application's needs.

An example of this configuration is writing new trusted handlers to control access to important peripherals. Besides creating trusted handlers, the developer also has to change the IoT application's code so it uses these handlers to interact with the peripherals instead of accessing them directly.

To the best of our knowledge, the use of CIDER, Lazarus, or something similar to our solution has not been implemented in a real-world scenario, at least from the documents that are accessible to the public. There are at least two reasons for this:

- Different Original Equipment Manufacturer (OEM) has different peripherals and TEE Implementation. Since the solution is an abstraction layer between the peripherals and the upper-layer application, it needs to have a standardized way to communicate in both directions (to the peripherals and the application). The same argument goes with the TEE.
- Not all OEMs open-sourced their operating system to the public. Unless the solution is standardized as a requirement for the device manufacturer, each OEM will have different approaches (and implementations) to solve issues raised in CIDER, Lazarus, and our paper.

11 Conclusion

This paper introduces an enhanced and more resilient approach to the remote recovery protocols outlined in CIDER and Lazarus. Specifically, we have restructured the procedures to ensure robust handling of network and runtime attacks.

To accomplish this, we introduce an independent `boot_mode`, LZ_VERIFY, which operates separately from the primary application. Its main function is to discern whether any disruption is due to an internal issue, such as a runtime attack, or an external issue, such as a network attack.

Using this method, the proposed design can better judge the device's status, hence enhancing the decision-making process for subsequent actions. This differs significantly from the Lazarus/CIDER approach, where both types of disruptions are treated similarly, resulting in the device becoming inoperable in case of a runtime attack, even if it is indeed possible, as we have shown in our design, to recover from such an attack. Through our approach, it is also possible to recover better from network attacks as they can be properly identified. Even if full recovery is not possible in this case unless a different network connection can be used, the device can go into minimal operation if the network attack persists.

Moreover, we have equipped the hub with the capability to directly manage IoT devices at any point when a `DeferralTicket` is received. In combination with the device state handling, this allows the hub to take a more proactive role in managing the device's lifecycle, given any environmental inputs. This is particularly essential for mitigating the effects of increasingly sophisticated runtime attacks.

We have shown that our design works by implementing it on the same platform as the Lazarus solution. Even if it is less straightforward, the same design can also be used with a hardware AWDT as used in CIDER. This requires more advanced ticket verification in hardware, which makes ticket control command updates more difficult, but apart from this, it is another possible realization of the proposed design.

Future works include implementing LZ_VERIFY in a secured part of the memory. This would need an evaluation board with a larger memory and prob-

ably port the current implementation with new peripherals. A work towards a distributed hub would also be an interesting way forward. Particularly, **if** the hub is not as strong as our assumption, a way to secure the hub needs to be designed.

Acknowledgements

Work supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research, EU H2020 project ELASTIC under grant 825473, and in part by the Wallenberg AI, Autonomous Systems and Software Program (WASP). We also thank Martin Gunnarsson for the initial discussion of this work.

References

[AC15] W. Arthur and D. Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. 1st. USA: Apress, 2015.

[AC20] M. Ammar and B. Crispo. "Verify&Revive: Secure Detection and Recovery of Compromised Low-end Embedded Devices." In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. Austin, USA: Association for Computing Machinery, 2020, pp. 717–732.

[ACT20] M. Ammar, B. Crispo, and G. Tsudik. "SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices." In: *ACM 11th ICCPS*. 2020.

[ADD21] S. F. J. J. Ankergård, E. Dushku, and N. Dragoni. "State-of-the-Art Software-Based Remote Attestation: Opportunities and Open Issues for Internet of Things." In: *Sensors* 21.5 (2021).

[ALK20] J. Ahn, I.-G. Lee, and M. Kim. "Design and Implementation of Hardware-Based Remote Attestation for a Secure Internet of Things." In: *Wirel. Pers. Commun.* 114.1 (Sept. 2020), pp. 295–327.

[Amm+19] M. Ammar et al. "SμV—The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things." In: *IEEE Transactions on Dependable and Secure Computing* 16.5 (2019), pp. 885–901.

[Ant+17b] M. Antonakakis et al. "Understanding the mirai botnet." In: *26th USENIX security symposium 2017*. 2017, pp. 1093–1110.

[Ati23b] S. A. Atiiq. *Improved Lazarus*. https://github.com/syafiq/lazarus-ng. 2023.

[Ati23c] S. A. Atiiq. *Network Attack POC against Lazarus*. https://github.com/syafiq/network-poc-lazarus. 2023.

[Ati23d] S. A. Atiiq. *Runtime Attack POC CVE-2021-31571*. https://github.com/syafiq/runtime-poc-lazarus. 2023.

[CRN23] A. Caulfield, N. Rattanavipanon, and I. D. O. Nunes. "ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation." In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5827–5844.

[De +22] I. De Oliveira Nunes et al. "CASU: Compromise Avoidance via Secure Update for Low-End Embedded Systems." In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '22. San Diego, California: Association for Computing Machinery, 2022.

[De +24] P. De Vaere et al. "The SA4P Framework: Sensing and Actuation as a Privilege." In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 873–885.

[Fra22] Fraunhofer-AISEC. *Lazarus: Healing Compromised Devices in the Internet of Small Things*. https://github.com/Fraunhofer-AISEC/lazarus. 2022.

[GRB03] T. Garfinkel, M. Rosenblum, and D. Boneh. "Flexible {OS} Support and Applications for Trusted Computing." In: *HotOS IX*. 2003.

[Gri+22] M. Grisafi et al. "PISTIS: Trusted computing architecture for low-end embedded systems." In: *Proc. of USENIX Security*. 2022.

[Gro19] T. C. Group. *TPM 2.0 Library*. Nov. 2019.

[Gro21] T. C. Group. *DICE Attestation Architecture*. Mar. 2021.

[Hri+18] S. Hristozov et al. "Practical runtime attestation for tiny IoT devices." In: *NDSS Workshop on Decentralized IoT Security and Standards (DISS)*. Vol. 18. 2018.

[Hub+20] M. Huber et al. "The Lazarus Effect: Healing Compromised Devices in the Internet of Small Things." In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 6–19.

[IST19] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik. "HEALED: HEaling & Attestation for Low-End Embedded Devices." In: *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2019, pp. 627–645.

[JKT23] S. Jakkamsetti, Y. Kim, and G. Tsudik. "Caveat (IoT) Emptor: Towards Transparency of IoT Device Presence." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1347–1361.

[JPF17] L. Jäger, R. Petri, and A. Fuchs. "Rolling DICE: Lightweight Remote Attestation for COTS IoT Hardware." In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. Calabria, Italy, 2017.

[LJF24] D. Lorych, L. Jäger, and A. Fuchs. "Acceleration of DICE Key Generation using Key Caching." In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*. ARES '24. Vienna, Austria: Association for Computing Machinery, 2024.

[Mae+18] P. Maene et al. "Hardware-Based Trusted Computing Architectures for Isolation and Attestation." In: *IEEE Transactions on Computers* 67 (2018).

[MWW02] B. E. Moret, L. Wang, and T. Warnow. "Toward New Software for Computational Phylogenetics." In: *Computer* 36.07 (July 2002), pp. 55–64.

[Nun+19] I. D. O. Nunes et al. "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation." In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1429–1446.

[Nun+20] I. D. O. Nunes et al. "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 771–788.

[Pet+22] L. Petzi et al. "SCRAPS: Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier." In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3485–3501.

[Pro+19] C. Profentzas et al. "Performance of Secure Boot in Embedded Systems." In: *DCOSS 2019*. 2019, pp. 198–204.

[RBM24] J. Röckl, N. Bernsdorf, and T. Müller. "TeeFilter: High-Assurance Network Filtering Engine for High-End IoT and Edge Devices based on TEEs." In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 1568–1583.

[Röc+21] J. Röckl et al. "Advanced System Resiliency Based on Virtualization Techniques for IoT Devices." In: *Proceedings of the 37th Annual Computer Security Applications Conference*. ACSAC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 455–467.

[Sai+04] R. Sailer et al. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004.

[Sem19] N. Semiconductor. *LPCXpresso55S69 Development Board*. 2019.

[SL19] R. V. Steiner and E. C. Lupu. "Towards more practical software-based attestation." In: *Comput. Networks* 149 (2019), pp. 43–55.

[Sur+21] S. Surminski et al. "RealSWATT: Remote Software-based Attestation for Embedded Devices under Realtime Constraints." In: *ACM CCS'21*. 2021.

[Suz+20] K. Suzaki et al. "Reboot-Oriented IoT: Life Cycle Management in Trusted Execution Environment for Disposable IoT devices." In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. Austin, USA: Association for Computing Machinery, 2020, pp. 428–441.

[Sys22] E. Systems. *ESP8266*. `https://www.espressif.com/en/products/socs/esp8266`. 2022.

[Tao+21] Z. Tao et al. "{DICE*}: A Formally Verified Implementation of {DICE} Measured Boot." In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1091–1107.

[Vli+19] J. Vliegen et al. "A Novel FPGA Architecture and Protocol for the Self-attestation of Configurable Hardware." In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 405.

[Wil+22] P. Williams et al. "A survey on security in internet of things with a focus on the impact of emerging technologies." In: *Internet of Things* 19 (2022), p. 100564.

[Xu+19] M. Xu et al. “Dominance as a New Trusted Computing Primitive for the Internet of Things.” In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1415–1430.

Attacks Against Mobility Prediction in 5G Networks

Abstract

The 5^{th} generation of mobile networks introduces a new Network Function (NF) that was not present in previous generations, namely the Network Data Analytics Function (NWDAF). Its primary objective is to provide advanced analytics services to various entities within the network and also towards external application services in the 5G ecosystem. One of the key use cases of NWDAF is mobility trajectory prediction, which aims to accurately support efficient mobility management of User Equipment (UE) in the network by allocating "just in time" necessary network resources. In this paper, we show that there are potential mobility attacks that can compromise the accuracy of these predictions. In a semi-realistic scenario with 10,000 subscribers, we demonstrate that an adversary equipped with the ability to hijack cellular mobile devices and clone them can significantly reduce the prediction accuracy from 75% to 40% using just 100 adversarial UEs. While a defense mechanism largely depends on the attack and the mobility types in a particular area, we prove that a basic KMeans clustering is effective in distinguishing legitimate and adversarial UEs.

Syafiq Al Atiiq, Yachao Yuan, Christian Gehrmann, Jakob Sternby, Luis Barriga. "Attacks Against Mobility Prediction in 5G Networks". In 22^{nd} *International Conference on Trust, Security, and Privacy in Computing and Communications, TrustCom 2023, Exeter, United Kingdom*. pp. 1502-1511, IEEE.

1 Introduction

According to the recent Ericsson mobility report[1], 5G mobile networks will reach 5 billion subscriptions by 2028. Compared to the previous generations, 5G will become the main subscription type in the coming years due to its support for higher speeds, higher bandwidth, lower latencies, improved security/privacy, and network capabilities exposure towards industry verticals and enterprises. For this reason, 5G is also considered a platform that enables innovation to investigate novel data-driven use cases, applications, and automation. NWDAF is a new network function that the 3G Partnership Project (3GPP) formalized in response to the maturity of Artificial Intelligence (AI) in recent years. NWDAF intends to benefit from the explosion of data generated by 5G. Its architecture and API are documented in 3GPP Technical Specification (TS) 23.288 [3GP22c] and TS 29.520 [3GP22a], respectively. NWDAF's main objective is to provide internal analytics service to other NFs to automate processes, i.e., troubleshooting, resource allocation, and anomaly detection. Based on [3GP22c; 3GP22a], one of the use cases of NWDAF is UE mobility analytics and prediction. For this context, the prediction can be used to optimize the resource allocation to be as precise as possible based on the user's needs. This is to enhance user experience and efficiency, as edge computing resources for heavy applications need to be pre-allocated before the user moves into a particular area. TS 37.817 [3GP22b] mentioned that trajectory prediction can be used to reduce late/early or wrong handovers. Looking from the opposite angle, the same mobility prediction can be used to detect whether a particular UE mobility is possibly adversarial. The definition of adversarial UE mobility itself is broad, but we are particularly interested in the deliberate actions of UEs targeting the accuracy of a mobility model within NWDAF. Therefore, in our context, an adversarial UE is one that tries to fool such a model and ultimately degrade its utility. This paper investigates possible attacks against a mobility trajectory model in NWDAF. In a real-world scenario, the model may be implemented differently, depending on the UE's mobility type, coverage, and density in a particular area. Given a set of legitimate UE movements, we would like to understand how the malicious activity of adversarial UEs could influence the prediction of the mobility model implemented in NWDAF. Since NWDAF is a new concept in 3GPP, attacks against its mobility trajectory models are yet to be found. While some of our attacks are hypothetical, based on the assumption that the adversary has control over a number of UEs, others are based on similar, earlier attempted attacks against the existing system. As a presumption, our work does not consider outliers in the network, i.e., certain events that gather large crowds. We experimentally evaluated the best model to be used for legitimate mobility, then attacked the same model with the intention of decreasing its prediction accuracy. To the best of our knowledge, this is the first work to investigate and evaluate attacks against mobility prediction models in NWDAF. As a follow-up, we investigate

[1] https://www.ericsson.com/en/reports-and-papers/mobility-report

how to differentiate the adversarial UE movements from the legitimate UEs. This can be used to prevent the mobile operator from including the adversarial UEs in the next retraining period so that the model degradation quality can be avoided.

The paper is organized as follows. We overview the problem definition in Section 2. We manifest the UE mobility in Section 3. The simulation environment and the results are written in sections 4 and 5, respectively. Defense mechanisms for the respective attacks are discussed in 6. We present the related work in section 7 and finally draw the conclusion in section 8.

2 Problem Definition

NWDAF collects data streams from neighboring NFs and performs analysis on the streams to support NF's tasks through an NWDAF subscription model. In this paper, we investigate adversarial mobility patterns and how they can influence NWDAF mobility models, which in turn will impact operations for the NFs that utilize these models. The mobility data is usually streamed from the Mobility Management Entity - MME (for 4G) or Access and Mobility Management Function - AMF (for 5G). In this work, we do not differentiate mobility data by origin; therefore, the base station in the system, i.e., eNodeB (4G) and gNodeB (5G), will be used interchangeably.

2.1 Scenario

We consider the NWDAF's use case of UE's mobility analysis and prediction. In our case, the output of mobility prediction is often consumed by the mobility management subsystem. It may also be used by its surrounding support services (for simplicity, let's call this an "application server"). Mobility predictions are useful to prepare the necessary network resources before the UE moves into another location. In a non-malicious setting, this will enable smooth handover and provide optimal data traffic routing [Jeo+21a]. Figure 1 depicts the scenario of a moving UE in a 5G setting. Initially, the UE is attached to some radio and Core Network (CN) NFs - gNodeB, Access and Mobility Management Function (AMF), Session Management Function (SMF), and User Plane Function (UPF) - commonly via the closest edge site.

At some point, the UE moves into a different geographical area where the currently allocated resources are sub-optimal. This move implies (i.) radio handover (changes of gNodeB), (ii) CN handover (AMF & UPF) re-selection, and (iii) the change of the application server (not shown in the picture). With the introduction of mobility prediction in NWDAF, the aim is to perform this mechanism automatically such that the target network resources at the destination are ready for the handover when the UE is moving. Practically, this will smoothen the process and provide a more efficient handover. For high-mobility UEs, mobility management manages network resource re-allocation in order to ensure service continuity and

QoS. This implies efficient handovers across radio and core network resources such as gNodeB, AMF, UPF, and Application Function (AF). Note that a radio handover between gNodeBs doesn't necessarily imply a handover between CN nodes such as UPF. That is dependent on the network configuration. Upon handovers, the network may need to spawn new resources on the destination and reduce the ones in the original location.

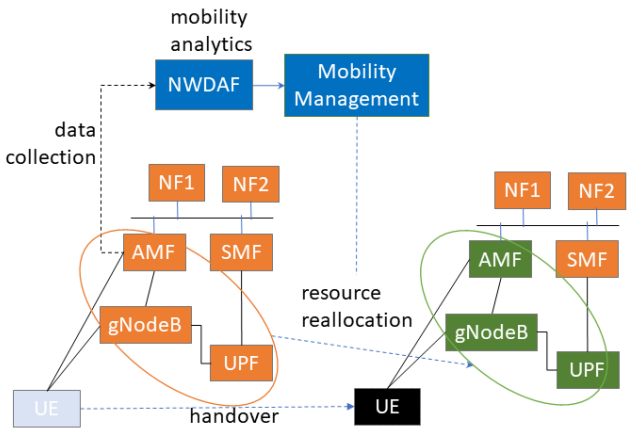

Figure 1: Use Case Scenario

Adversarial UEs aware of this scenario could try to induce the mobility trajectory model deployed in NWDAF with malicious movement, i.e., moving around the network with the intent of decreasing the accuracy of the prediction performance of the model. With reduced prediction performance, network resources could be wrongly allocated, causing inefficiency and decreased Quality of Service (QoS).

2.2 Threat Model

We assume an adversary has the power to acquire UEs and then clone [Liu+15a; Liu+15b; BFD20; Zha+21; P1 21] them to obtain a number of UEs with the same identity. Acquiring UE does not necessarily mean that the adversary needs to steal sim cards, but legitimately owning a sim card can also be counted [Wec20]. A practical example of this action is when a powerful adversary legally acquires a large amount of IoT devices in a short time. In this case, there is no need to hijack anything, and cloning the device is much easier as the source and cloned devices are already owned by the adversary.

Similar types of attacks might increase considerably in the future. For example, the 3GPP security report TR 33.861 [3GP19] identified two threats stemming from the massive deployment of cellular IoT devices in 5G, stating that a malicious attacker could take over the same application on a large number of UEs and

instruct them to launch signaling attacks at the same time. In our case, we argue that the UE application can be manipulated to generate malicious mobility patterns instead. The same document also mentions the risk of massive low-security unattended cellular IoT devices, such as shared bicycles, that can be hijacked and used to launch Denial of Service (DoS) attacks toward the radio network. In our case, we argue that the attack can be used to generate malicious mobility patterns instead. The Mirai botnet is an example of how the infection could also possibly spread across massive cellular IoT.

An adversary may increase the number of cloned UEs to many "perceived to be" same UEs by the operator. It has been shown that extracting the USIM (Universal Subscriber Identity Module) master secret key in 4G is feasible [Liu+15a; Liu+15b]. A more advanced side channel attack to compromise USIM has also been shown using deep learning [BFD20]. Compromising the master USIM key means that an adversary has the capability of cloning them. Aside from compromising the USIM, a temporal UE's cloning can also be achieved by intercepting the authentication vectors generated by the network and stealing them via a false base station[P1 21]. While this was proven to be successful against 2G, 3G, and 4G, the same issue will be faced by 5G interconnects if the operator does not enforce a coherency control on the inbound 5G signaling [P1 21].

In this work, we present several malicious mobilities, explained in more detail in section 3.4. The objective of such mobilities is to decrease the overall prediction accuracy of a mobility model. From the operator's perspective, a decreased accuracy would mean two things: (i.) This can lead the mobile operator to believe that there is a shift in UE mobility patterns, which may prompt them to retrain the model with newer data. (ii.) if the former is executed, then the prediction quality is significantly downgraded, which means the predictions of the model are unreliable. Consequently, the efficiency, as well as the resource allocation function of networks, would drastically deteriorate. Meanwhile, even if the defender can retrain the model periodically to include new patterns, retraining is not only time/resource-consuming but also potentially harmful to the model's performance on *normal* data.

3 Mobility

The uniqueness of human mobility to a particular location is a reflection of the culture and values of the people living there [ST19a]. Each location has its own characteristics, which influence how people move, the time they spend in a geographical area, and the motivations for their travels. That said, the best way to model human mobility is by using a real dataset obtained from real UE mobility in a live network.

Since there are, to our knowledge, no available UE mobility datasets collected by network operators, we decided to generate a synthetic one using a mobility simulator created by Ericsson [Eri20] using a publicly available mobile network

topology. This section explains in more detail the simulator, the legitimate mobilities, and the adversarial mobilities developed on top of the simulator.

3.1 Mobility Simulator

The mobility of a UE is sensitive data that has to be treated carefully. Hence, obtaining real data from a mobile operator is difficult due to privacy concerns governed by strict laws, i.e., General Data Protection Regulation (GDPR) in Europe [Eur23]. To overcome this, Ericsson has developed a spatiotemporal mobility simulator, producing the dataset of UE mobility given a set of pre-determined mobility models. In this work, even though the mobility data is based on a predetermined model, the spatial aspect of the simulator is based on a real-world deployment by Airtel's open-network topology [Air16] in India.

The dataset of UE mobility is a time series. For a particular time (let's call this a timestamp t), it manifests a UE connected to a certain eNodeB with a calculated signal strength depending on its distance and the load of the respective eNodeB. The original mobility simulator contains legitimate mobilities in section 3.2, while the extended version contains additional malicious mobilities from section 3.4. While the simulator is not open-source, a subset of the post-processed dataset from our work is available at the following repository [Ati23a].

3.2 Legitimate Mobilities

Accurately modeling human mobility patterns is a significant challenge due to their high variability across different locations worldwide. This variability makes it difficult to develop a generalizable approach for predicting movements at a specific time and place. In our study, we focus on a radio network installation in India (and in particular, Airtel [Air16]), with legitimate mobilities consisting of people who are working in an office setup (with a particular and consistent location of office and home), people who drive taxis, and local street vendors who move around in a limited particular area. While this is not an exhaustive list, we select representative mobility types that can be formalized and categorized into the same type, allowing us to make meaningful conclusions based on similar movement types.

Working Professional (WP)

Working Professional is a custom mobility model designed to simulate the pattern of a UE belonging to an employee with a stationary workplace. It follows a periodic movement pattern between the office and the home. Professionals are assumed to move between home and office during the day and evening. Initial coordinates for home and office are randomly selected (for a given constraint, i.e., specific boundaries on a particular location) when the mobility model is constructed. Lastly, the mobility pattern is assumed to happen on a daily basis. Such mobility has been

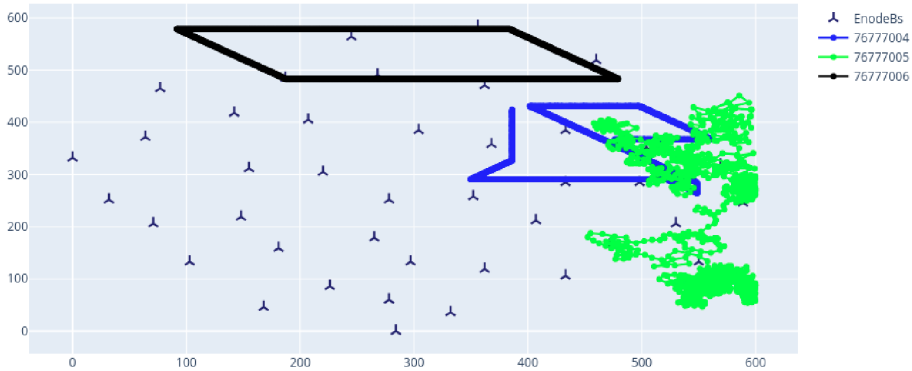

Figure 2: 3 UEs with different mobility. Black: Working Professional, Blue: Random Way Point, and Green: Gauss-Markov.

formalized in [Ekm+08] and represents the population of 140 million people in India[2].

Random Way Point (RWP)

The Random Waypoint [HV07] model simulates random UE mobility behavior over time. It randomly generates destination, speed, and direction for each UE, independently of other UE's, allowing the UE to move freely and without restrictions. This model has been used in numerous simulation studies [Deh+20; Ge+16; Nai+05]. The movement of a UE is governed in the following manner: Each UE may be able to move/pause for a timestep based on a probability value between 0 and 1. At a particular time, this indicates how likely a UE is going to move or not. At each step, a random destination, minimum speed, and maximum speed are selected. The UE moves from its current location to the randomly selected destination and repeats this process for the duration of the simulation. This mobility is analogous to a taxi driver, where the direction and the time spent at each point are uniformly random, depending on the request from the passenger. There are estimated to be around 1.9 million taxi drivers in India[3]. While they are still a fraction of the total population, the mobility behavior of taxi drivers is common in downtown areas (regardless of the country) in general. Another reason why we include RWP is that, unlike WP, which usually takes only one round-trip per day, a taxi driver often takes more than a trip, depending on how many customers they serve, generating a larger number of mobility events.

[2]https://www.thehindu.com/data/India-walks-to-work-Census/article60346511.ece
[3]https://mobilityforesights.com/product/on-demand-taxi-market-in-india/

Gauss-Markov (GM)

The Gauss-Markov model [AWS06] is a mobility model that enables objects to move across a grid plane, with their next transition point determined by their current speed and direction. One can think of this model as similar to a local vendor/salesperson walking around a particular area. There are estimated to be about 10 million street vendors in India[4].

Figure 2 shows the movement of WP (Black), RWP (Blue), and GM (Green) across a particular Airtel India's cartesian plane. For illustrative purposes, each mobility is shown with a single UE, while the actual simulation involved 10,000 UEs for each mobility.

While WP is intuitively the most predictable mobility among the three, the unpredictability of random mobility models such as RWP and GM presents a challenge for accurate prediction, as movement patterns are inherently random. To address this challenge, it may be more important to determine the proportion of unpredictable UEs relative to the total number of mobile subscribers in a given area. This information can be used to allocate appropriate resources for these unpredictable UEs. Despite the seemingly random nature of their movements, it is important to note that random UEs are still subject to the laws of physics, and therefore, it is impossible for a UE to instantaneously jump from one location to another. By accounting for the proportion of random UEs and the physical limitations of their movement, it may be possible to develop more effective strategies for managing network resources in areas with high levels of random mobility.

3.3 Prediction Accuracy

In the case of mobility prediction, there are two important metrics to look into:

1. Location prediction at a particular time, denoted as $\hat{l}$.
2. Timeslot prediction ($\hat{s}$), given that $\hat{l}$ is correct. This shows how long a UE stays in a particular enodeB at a particular time.

Supposed that there are n mobility events e for a period of $t_2 - t_1 \in T$, the mobility prediction accuracy is the sum of correct timeslot prediction $\hat{s}$ (given that the location prediction $\hat{l}$ is also correct), divided by the total number of predictions. Formally, the total accuracy can be written as follows:

$$\textbf{Accuracy} = \frac{\sum_{e=0}^{n}(\hat{s}_{correct}|\hat{l}_{correct})}{n} \tag{1}$$

This accuracy calculation from equation 1 will be used for the rest of this paper.

[4]https://www.indiaspend.com/governance/only-11-of-vulnerable-street-vendors-benefitted-from-pm-credit-scheme-survey-774968

3.4 Adversarial Mobility

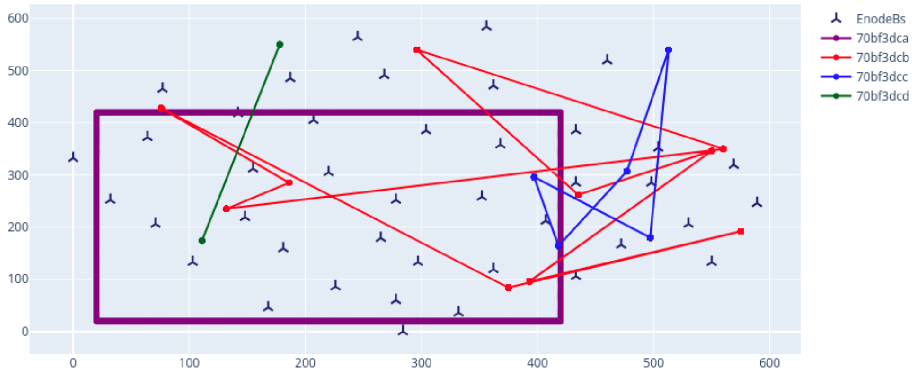

Figure 3: 3 UEs with different attack patterns. Green: Tuple Jump, Blue: Quintuple Jump, Red: Decuple Jump, and Purple: Google Maps Attack.

This section outlines the existence of adversarial UEs with the intention of reducing the accuracy of legitimate mobility patterns by introducing adversarial movements. Thus, shifting the perceived UE's behavior such that the operator feels the need to retrain the NWDAF's model with newer data (including adversarial UEs). The adversarial movements are required to be economically efficient by committing to the lowest possible resources while aiming for the highest decrease in accuracy. The resources available to the adversary include the number of acquired UEs and the performed adversarial mobility. The best-case scenario for the adversary is to produce adversarial movement without physically carrying around the UEs, as it minimizes the effort required. However, if the UEs have to be physically transported, the effort involved must be minimized to ensure the sustainability of the attack. In this scenario, an adversary, denoted as a, successfully acquires n UEs, and we aim to understand the optimal strategy for a to achieve the lowest accuracy as quickly as possible using the lowest resources in hand.

Tuple Jump Mobility Attack

Given that a acquired n UEs, a divides n into $n/2$ paired sets. A set contains two distinctively different UEs but has the same identity, i.e., International Mobile Subscriber Identity (IMSI). As the basis of our assumption, it has been proven that it is feasible to clone the UE, as explained in our threat model. We built the adversarial mobility on top of this argument.

Within a set, let's call the first UE u_1 and the second one u_2. Even though they are distinctively different, from the network operator standpoint, u_1 and u_2 are the same (due to the same identifier, IMSI). The following sequences illustrate the attack:

1. At timestamp t_1, u_1 connects to an enodeB e_1. At the same time, u_2 stays silent and does not perform any effort to connect to any enodeB.

2. At timestamp t_2, u_1 shuts down its interface, while at the same time, u_2 connects to another enodeB e_2 located on a different geographical location than e_1.

3. Repeat from the 1^{st} step.

What the network operator sees is a UE jumping around from e_1 to e_2 and back forever. The preference on which e_1 and e_2 should be connected is uniformly random, selected by adversaries. Not only e_{1-2} can be located adjacent to each other but also in a sparsely different location where movement from e_1 to e_2 can be impossible from the law of physics.

However, it is limited to the cartesian plane that we extract from Airtel's open-network [Air16]. Using this approach, a does not have to carry around u_1 and u_2 to make both of them look like they are moving from e_1 to e_2. The effort to perform the mobility on this attack is virtually non-existent as u_1 and u_2 can be statically placed near e_1 and e_2, respectively. We consider the effort to put u_{1-2} in the first place to be negligible, as it is only needed when a deploys the adversarial UEs, not when operating them.

Quintuple Jump Mobility Attack

While in the previous attack model, a divides n into $n/2$, in this scenario, a divides n into $n/5$, expanding the tuple $[u_1, u_2]$ into a quintuple $[u_1, u_2, u_3, u_4, u_5]$. We would like to understand whether expanding the tuple has any positive effect on the decreasing accuracy or not. A set contains UEs with a cloned identity; hence they are perceived to be the same UE by the mobile operator. The sequences are similar. But, instead of repeating in step number 3, the jumping traverses through all the members u_{1-5} and repeats the sequence in step 6. Similar to the previous one, the sets can be vertically scaled up depending on the power of a.

Decuple Jump Mobility Attack

Similar to the two previous attacks, the number of acquired UEs n are divided into sets, where one set is a decuple, consisting of $[u_{1-10}]$. These sets can be vertically scaled up, depending on the power of a.

Google Maps Attack

An adversary a acquires a set of n UEs and walks a particular path while carrying them. This type of attack has similarities to a demonstrated attack against Google Maps, where a attempts to create a fake traffic jam in the application by carrying UEs in a basket [Wec20]. We aim to examine whether this attack can be applied

to the mobility trajectory in NWDAF. Additionally, we explore the scalability of this attack by varying the number of acquired UEs.

Figure 3 shows the movement of tuple jump, quintuple jump, decuple jump, and google maps attack in green, blue, red, and purple, respectively. Similar to the picture of legitimate mobilities, we only show the movement of each mobility with one UE, spanned over two days of simulation.

4 Simulation Environment

The simulation of UE mobility was conducted on a server equipped with 40 Intel Xeon E5-2650 cores and 128 GB memory. In the baseline simulation (i.e., without attack), 10,000 UEs were included in each of the legitimate mobility, WP, RWP, and GM scenarios. The simulation spanned five days, with the first four days allocated for training and the 5^{th} day used for testing.

On top of that, the actual model used in this simulation is not part of the NWDAF specification, as the 3GPP standard only specifies interfaces and use cases. That said, the model that we developed here, together with the follow-up attack scenario, is an attack against a mobility model with input and output as if it were deployed in NWDAF.

One might argue that our research should have used real-world data instead of a simulated one. However, obtaining real-world data for the use of NWDAF is not as simple as it sounds. Regulation-wise, the data is considered to be sensitive. While anonymization might help, the operators will not give an interested 3^{rd} party access to such data. Our work is an initial effort to open up the research direction in the area despite these limitations. We simulate the legitimate mobilities in a way that it should be enough that there are different mobilities, following human mobilities from the previous research [ST19a]. It allows the needed amount of data (in order to get significant and reliable experimental results) to be generated under different mobility patterns, reflecting the reality that mobile data is heterogeneous over time and at different geographical locations. Although the used mobility data is synthetic and will differ somewhat from data of a real deployment, we believe that this does not affect the validity of evaluating the threats. The investigated attacks create mobility patterns that differ from the mobility models, and although results would differ for models built from real-world data, the principle would also be applicable to real scenarios.

5 Result

The 3GPP standard does not explicitly specify whether one or more models should be used to perform trajectory prediction. In this study, we utilize a different model for each legitimate mobility type. This approach is motivated by two factors. Firstly, certain mobility types are easier to predict than others, as evidenced by

the WP mobility (see table 2), which outperformed the other mobility types. Secondly, combining the datasets into a single training and testing set would not make sense since the poor performance of some mobility types would overshadow the good results of others.

We assume that the mobile operator knows the mobility type of each UE before the adversarial UEs are introduced by the attacker. This can be accomplished by examining the UE's history and attaching a label to each IMSI. Alternatively, unsupervised learning can be used to cluster the results into a labeled dataset with mobility types attached to each UE. While previous studies have proposed spatiotemporal clustering techniques for this purpose [Lu+21b][Kis+10], our work does not include these techniques.

5.1 Data Preprocessing

The dataset used in this study, generated by [Eri20], includes three core features: {IMSI, enodeb_path, signal_strength}. Each unique IMSI serves as an index, with the historical data of visited eNodeBs, their corresponding timestamps and signal strengths recorded in each data point. Take the following example of a raw data point:

IMSI	enodeb_path	signal_strength
09ccc864	'163':12,'348':10,...	'163':0.189,'348':0.186,...

Table 1: An example of a raw, IMSI-based data point

This shows a UE with an IMSI `09ccc864` that connects to an eNodeB 12 at timestamp 163 with a signal strength of 0.189, followed by connecting to eNodeB 10 at timestamp 348 with a signal strength of 0.186, and so on. To perform a timewise prediction, where the NWDAF predicts which eNodeB a UE will connect to and for how long it will stay connected at a particular time, the dataset is transformed from IMSI-based to an event-based. An event occurs when a UE connects to an eNodeB with one timestamp representing one minute. The duration of time that a UE stays connected to a particular eNodeB is calculated as the time elapsed between the current and the next connection to another eNodeB. This calculation is used to create the timeslot column for each event, providing more complete information. Given the example above, a timeslot with the value of 185 (coming from 348-163) is added to the timeslot column of the data point.

To contextualize the dataset, we incorporate historical information by appending the values of the previous four connected eNodeBs and their respective timestamps (relative to each day) to each data point. Additionally, we include the top four neighboring eNodeBs based on the adjacency matrix of Airtel's eNodeB network installation [Air16] to provide an additional location context. This information enables us to determine the top four nearest eNodeBs available for a

	WP	RWP	GM
Accuracy	75.2%	44.8%	37.6%

Table 2: Baseline accuracy for each mobility type

given target eNodeB location. We categorize the timestamp of each IMSI's mobility into five distinct time bins: {early_morning, morning, noon, evening, and night}, and then aggregate the sum of each bin to provide information on how each IMSI spends its time on a particular day. Finally, we also include the eNodeB to which a UE stay connected the longest, called home_enodeb. The final input variable (x) includes the following enriched features:

```
{enode_1, enode_2, enode_3, enode_4, time_1, time_2
    , time_3, time_4, target_time, sig_st, imsi,
    home_enb, early_morn, morning, noon, evening,
    night, neigh_1, neigh_2, neigh_3, neigh_4}
```

The dataset is used to predict a binned timeslot ($\hat{s}$) that a UE stays connected to a particular eNodeB ($\hat{l}$) using the model discussed in Section 5.2.

5.2 Baseline

Table 2 presents the baseline accuracy of each mobility type when there are no adversaries. The results indicate that the mobility type of working professionals has the highest accuracy since it is the most predictable. Conversely, the other mobility types perform poorly due to their randomness in destination, speed, and direction (for the random waypoint) and a random variable sampled from the Gaussian distribution (for the Gaussian-Markov).

Each event previously mentioned in section 5.1 is independently treated to produce two predictions (namely $\hat{s}$ and $\hat{l}$), i.e., the models have been trained independently of each other. The baseline accuracy presented in Table 2 is obtained by running the classifier in Table 3 on the dataset previously pre-processed, with the classifier and its hyperparameters automatically selected by the FLAML AutoML framework [Wan+21a]. We did not manually choose the classifier or its hyperparameters.

5.3 Adversaries Included

Figures 4, 5, and 6 demonstrate the impact of adversarial mobility on the total mobility prediction accuracy. Each graph shows the changes in accuracy for the different legitimate mobility types as the number of acquired adversarial UEs increases. The dataset was expanded by the data points coming from the adversarial UEs when the attack occurred. After preprocessing, there were no noticeable

Mobility	Model $\hat{l}$	Model $\hat{s}$
WP	Random Forest [Bre01]	Extra Tree [GEW06]
RWP	Random Forest [Bre01]	Extra Tree [GEW06]
GM	Light GBM [Ke+17]	Extra Tree [GEW06]

Table 3: Model used for each mobility type

changes (decrease or increase) in the accuracy value if we only selected the legitimate data points. However, the accuracy value decreased due to the addition of events with low accuracy from the adversarial UEs. We did not observe any connection between the introduction of the adversarial UEs and the accuracy value of legitimate UEs. This is because the preprocessing mechanism, as explained in section 5.1, does not involve mixing inter-IMSI data when we add historical context to each data point. Nevertheless, NWDAF treats all UEs as a legitimate source of data, as it is unaware of the existence of adversarial UEs at this point. In short, by introducing adversarial events, the mean accuracy will go down by the proportion of adversarial events.

When n UEs are acquired, the optimal strategy for the adversary is to execute the tuple jump attack. This involves dividing the attacker UEs into $n/2$ pairs of cloned IMSIs, maximizing the decrease in accuracy. The quintuple and decuple jump attacks also decrease the accuracy number, but not as fast as the tuple jump attack.

The tuple jump attack stands out as the most effective because it generates a greater number of data points compared to other attacks. For instance, given $T \ni t_0 - t_{10}$ and 10 acquired UEs by a, at t_{10}, the generated data points are $\{50, 20, 10\}$ for {tuple, quintuple, decuple}. Moreover, at any timestamp, the tuple jump attack has only one idle UE, while the quintuple and decuple jump attacks have four and nine idle UEs, respectively. While having the idle UE is necessary due to the unwillingness of the adversary to physically carry them around, optimizing their existence is necessary, and one way to do that is by minimizing the number of idle UEs.

Although the various jump attacks have different effects on accuracy, the Google Maps attack appears to have little impact on overall accuracy. Graphs for WP and GM indicate a slight decrease in accuracy as the number of acquired UEs increases, but the opposite occurs for RWP, where accuracy actually slightly improves. One possible explanation for this phenomenon could be attributed to the intrinsic randomness of the RWP mobility model, which makes it challenging to develop an accurate prediction model. In contrast, the Google Maps attack involved a set of devices with more predictable movements. Strikingly, as the proportion of predictable devices increased in the dataset, we observed a paradoxical effect: the accuracy of the prediction model improved instead of decreasing, as one

might expect. Therefore, an adversary seeking to quickly decrease accuracy would not need to use this method, at least not against an NWDAF model with the UEs comprised of WP, RWP, and GM mobility.

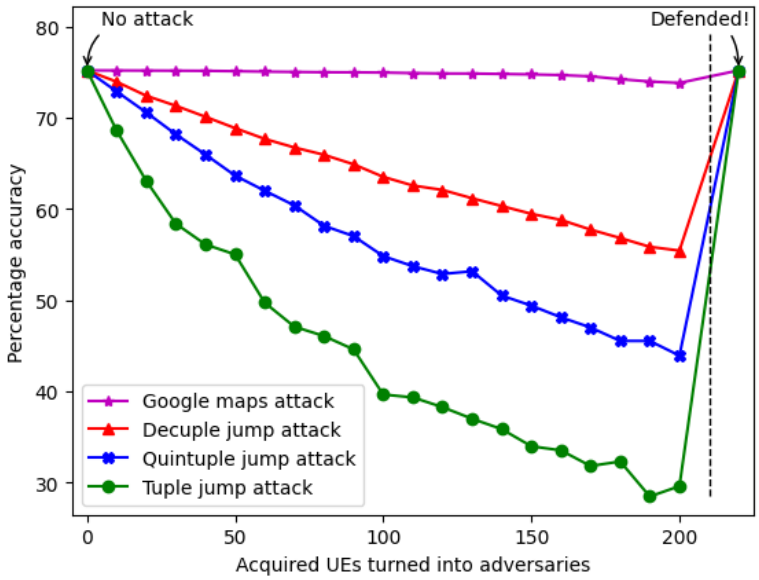

Figure 4: The number of acquired adversarial UEs on the accuracy of WP's mobility

6 Defense Mechanism of The Deployed Model

In light of the attacks discussed in the previous section, safeguarding the trustworthiness of legitimate users is paramount. Although the accuracy of legitimate UE may remain unchanged, it is imperative to defend against a decrease in total accuracy (legitimate+adversarial) since a general mobility prediction model is unable to distinguish adversarial data.

As a way to analyze the properties of the adversarial data, i.e., how easy it is to distinguish the adversarial from the legitimate data, we employ KMeans [HW79]. It is chosen due to its simplicity and the most popular clustering methods. The adversarial samples in our dataset are distinct from normal samples, making them easy to cluster using KMeans. While KMeans is not intended for real-time detection of adversarial UEs, it can be used to solve the problem of malicious mobility patterns by identifying which data is safe or unsafe to be included in the retraining process. This is because the main problem, as explained in Section 2.2, is that a shift in UE behavior can lead the mobile operator to believe that retraining with newer data is necessary.

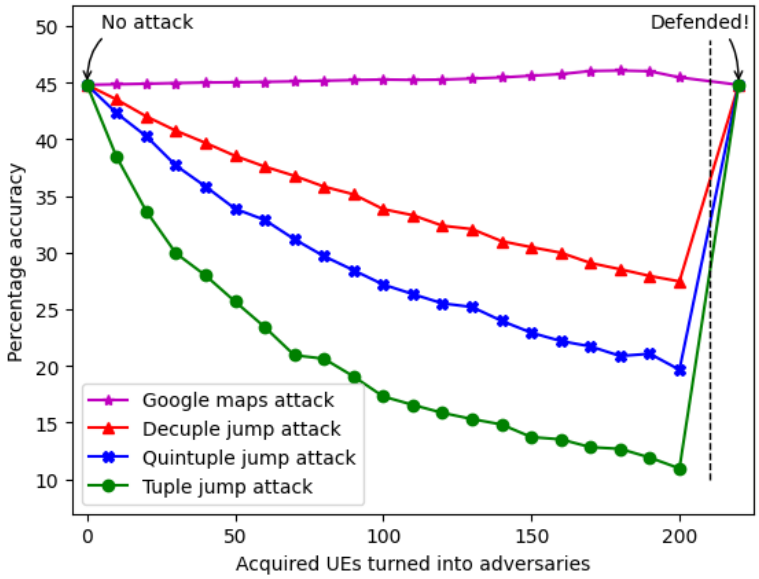

Figure 5: The number of acquired adversarial UEs on the accuracy of RWP's mobility

Although there may be other methods to defend against adversarial UEs, such as increasing the robustness of the deployed model, we leave this as an open question. We believe that more attacks will be introduced in the future, and the solution will depend on the respective attacks. However, for adversaries that fall within the constraints defined in our threat model, KMeans serves as an adequate distinguisher.

Given a set of UE location data $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$, $x_N \in \mathbf{R}^d$, where $\mathbf{x}_i$ has three features, i.e., {IMSI, enodeb_path, signal_strength}, KMeans splits the dataset that includes both malicious and benign samples into K disjoint sub-clusters $C_1, ..., C_K$. It aims to minimize the distance between a data point and the centroid of every sub-cluster. That is to say, the data sample belongs to the closest sub-cluster. Then, we calculate the hamming distance of the accuracy using each of the clustered labels. Equation 2 is the clustering loss where f_i is the clustering loss, x_i is the i^{th} sample and o is the cluster center. $d(x)$ is the hamming distance function, N is the total number of samples in a dataset. In our settings, we set $K = 2$ for malicious and benign clusters.

$$f_i = \sum_{k=1}^{K} \sum_{i=1}^{N} g(x_i \in C_K)[d(x_i, o)]^2, \tag{2}$$

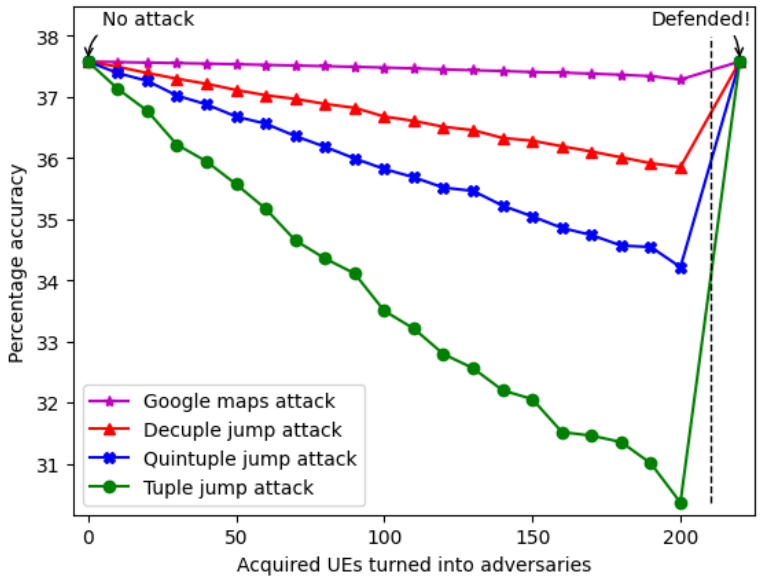

Figure 6: The number of acquired adversarial UEs on the accuracy of GM's mobility

6.1 Dataset

To evaluate the effectiveness of KMeans, we analyze its performance using three different datasets that pertain to WP, RWP, and GM mobility patterns, where the WP, RWP, and GM datasets contain 110811, 80930, and 613306 normal data points generated by 10000 UEs. Besides, the WP, RWP, and GM datasets include 233949 adversarial data samples, i.e., 142560, 56736, 27792, and 6861 from Tuple Jump, Quintuple Jump, Decuple Jump, and Google Maps attacks. The data points of each attack model are produced by 200 UEs (the highest number of acquired UEs from figures 4, 5, and 6). In total, there are 805047 legitimate and 701847 malicious samples in our dataset. In our experiments, the parameter n_clusters=2 as we have two classes, malicious and benign; as for other parameters, we use the defaults of the KMeans() function in *sklearn*, e.g., random_state=0, n_init=10.

6.2 Results

The visualization result of KMeans clustering for GM's mobility and the Tuple Jump mobility attack is presented in Figure 7. The figure displays the timestamps and timeslots of one legitimate and one adversarial UE from a specific timestamp range on the 5^{th} day of the simulation. The timeslot is defined as the time duration of a UE staying connected to a specific eNodeB, which is calculated by the elapsed

time between the current and the next connection of the UE to another eNodeB. The clustering result demonstrates that KMeans successfully distinguishes between the timeslots of legitimate and adversarial UEs. The legitimate UE has a varied timeslot for each connection, while the Tuple Jump attack has a consistent timeslot throughout the simulation. Although an attacker may randomize the timeslot value to evade the clustering, a lower timeslot means more data points on the attack dataset, which will result in a much faster accuracy reduction. Thus, a more sophisticated adversary must strike a balance between having as many data points as possible and randomizing the timeslot value. We left this as an open question for future research.

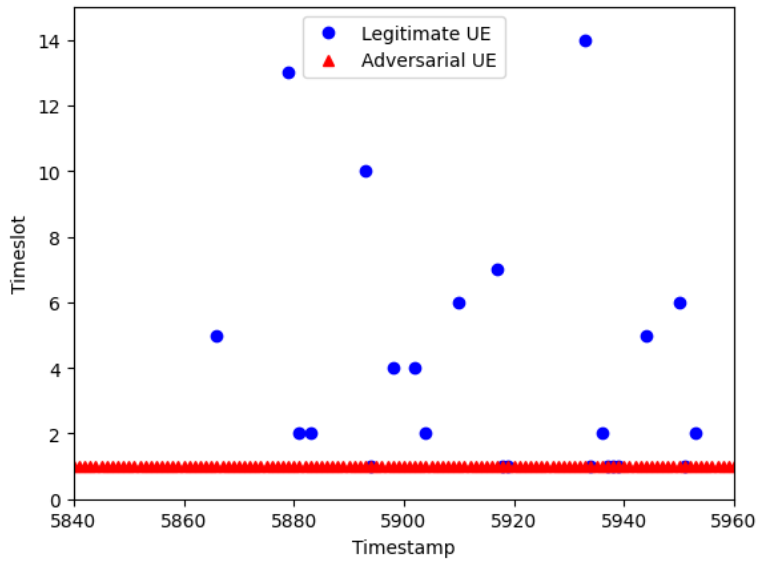

Figure 7: KMeans visualization of a selected timestamp on Gauss-Markov Mobility

Figures 4, 5, and 6 demonstrate the activation of the defense mechanism at their respective rightmost ends. When the defense mechanism is activated, the adversarial UEs can be separated from the rest of the dataset. It deters the mobile operator from retraining the model with newer data, as this could adversely affect the model's performance. The operator can then take follow-up actions on the separated dataset of adversarial UEs, such as blocking them or taking other measures to mitigate the attack. However, the specific follow-up actions are out of the scope of this paper.

7 Related Work

NWDAF is a critical component in 5G networks that enables network automation, optimization, and security, and there has been a growing interest in using NWDAF to enhance data analytics capabilities and enable more advanced automation and optimization of 5G networks. To enhance NWDAF's performance, numerous research studies have been carried out. They can be grouped into three categories, i.e., "NWDAF-oriented mobility trajectory prediction", "adversarial attack on network mobility", and "attack defense mechanisms".

NWDAF-oriented mobility trajectory prediction: In [Kim+22], the authors emphasized the significance of NWDAF for enabling real-time data analytics and decision-making in mobile networks and explained that NWDAF plays a crucial role in collecting, processing, and analyzing network data. [CMS22] delves into the integration and implementation of NWDAF in 5G/6G core networks. They discuss the architecture of NWDAF and its capability to provide slice-specific network data analytics to various NFs. Through a case study, the paper illustrates the insights obtained from NWDAF-collected data and highlights its potential in steering traffic policies and resource allocation, including from the mobility data.

Similarly, [Sha+20] highlighted the importance of utilizing machine learning algorithms in NWDAF for beamformer design and predicting mobility for better handover management. Mobility in the 5G core network plays a pivotal role in ensuring efficient and agile operations, as explained in [Jeo+21b]. The use of machine learning has shown potential in accurately predicting these mobility patterns within the core network domain. However, a significant challenge emerges in the practical application of these predictions. One notable limitation to consider is the sensitivity of the prediction models to changes in the network topology; for instance, the entire Recurrent Neural Network (RNN) model might require retraining if there is an addition or removal of a gNB. Despite these challenges, advanced predictions offer practical advantages: if the network can anticipate a UE move to a target edge site, it allows for preemptive actions in the relocation procedure, streamlining the process before the actual gNB handover occurs.

Adversarial attack on network mobility: As aforementioned, NWDAF is critical for automating data collection and analytics of the core network, and ML methods play a key role in NWDAF. However, due to the distributed and API-based structure, the ML algorithms in NWDAF could be possible targets for adversarial attacks [Ash+22]. Attacks on the network mobility can occur either during the training or testing phases. Training phase attacks, such as poisoning attacks, aim to corrupt the model by injecting manipulation into the training process that causes the trained model to make incorrect or biased predictions. For example, [Zol+23] considered a poisoning attack in 5G, where compromised UEs are utilized to transfer malicious data to RAN after using their credentials to authenticate and be authorized in the core network. Testing phase attacks attempt to create input data that appears normal but is specifically designed to mislead the machine

learning model to give the wrong results [Sze+14]. Our work falls into the category of testing phase attacks, specifically targeting the mobility trajectory model after it is deployed.

Attack defense mechanisms: There are mainly two types of defense mechanisms, proactive and reactive. Proactive defenders prevent adversarial attacks before they happen by, for instance, Moving Target Defense (MTD) [Sou+21] and distillation [Pap+16; Gro+17] and adversarial training [Ant+21], while reactive defenders respond to adversarial attacks when they occur, for example, using anomaly detection [STL20; MTB23; LTZ08]. For proactive defenses, MTD [Sou+21] defends against the attack by continuously changing networks' attack surface to make it more difficult for attackers to identify and exploit vulnerabilities. By using MTD, it is more challenging for attackers (by reducing their asymmetrical advantage) to find the vulnerabilities of the network, while the network defender can respond to attacks in real-time, and the network can adapt to changing threats.

Distillation as a defense mechanism [Pap+16; Gro+17] involves training both a "teacher" and a "student" model. The "teacher" produces softened output probabilities, which act as labels when training the "student". The idea is to have the "student" model learn the underlying patterns in the data rather than merely memorizing the training set, which can make it more robust to adversarial attacks. While distillation-based defenses can enhance resilience against certain adversarial attacks by promoting more robust feature learning, they are not universally foolproof [CW17]. In terms of computational efficiency, once the teacher model is trained, only the student model needs additional training, which can be more efficient than other defense methods that require multiple retrainings. Adversarial training is used to improve the robustness of a model against adversarial attacks by incorporating malicious data into the training set to improve the robustness of the model against future attacks. Yet, the computational cost could be increasingly high as it requires constantly generating and including adversarial samples for training. For instance, [Ant+21] evaluates the performance of a supervised-learning-based industrial control system when it is under adversarial attacks and utilizes the generated adversarial samples to improve the robustness of the system.

For reactive defenses, anomaly detection distinguishes the benign and malicious data. Drawing inspiration from the Isolation Forest algorithm [LTZ08], which is a previous method for outlier detection, [MTB23] extends its concept to a proximity-based framework. Unlike the Isolation Forest, which operates on feature-based outlier detection, the Proximity Isolation Forest requires only a set of pairwise distances to function, making it adaptable to different types of data. Anomaly Detection Forest [STL20] discusses the challenge of anomaly detection in a case where labeled anomaly data is not available and proposes a new algorithm designed for one-class learning with only normal instances as training data. The algorithm is an ensemble of binary trees and shows superiority over existing algorithms, i.e., Isolation Forest and One-Class Random Forest for one-class anomaly detection. While the purpose is not to perform a real-time classification, our work

in this paper is considered a reactive defense, where we employ KMeans to distinguish the adversarial UEs from the legitimate ones.

8 Conclusion

This study presents an investigation of potential attacks against mobility trajectory models supporting 3GPP NWDAF abnormal mobility normative analytics use cases. The results show that, under the constraint of an adversary who aims to minimize their effort along with the power of IMSI cloning, a tuple jump attack is the most effective strategy. This attack involves dividing the acquired adversarial UEs into pairs of cloned IMSIs, resulting in a decrease in mobility prediction accuracy from 75% to 40%, given 10,000 legitimate subscribers and 100 acquired adversarial UEs. The purpose of such an attack is to make the mobile operator believe that there is a shift in the UE's mobility behavior, and retraining with newer data is necessary, which leads to a lower accuracy on the legitimate UEs if this happens. While our attack works on a given legitimate mobility, i.e., working professional, random-waypoint, and gaussian-markov, further study is needed for different mobility types. Our analysis shows that the data from adversarial movements is distinct, and KMeans Lloyd's unsupervised clustering technique is sufficient to distinguish between legitimate and adversarial movements.

Future research directions include i.) testing the approach on a live network dataset and investigating the implications of retraining the model with adversarial data, ii.) modeling more sophisticated adversarial UEs, i.e., armed with the power to randomize the timeslot when jumping between points in the cartesian plane. This would need a more advanced defense strategy, as we would expect. And lastly, iii.) Expanding the investigation to include cellular IoT scenarios, which are expected to dominate the amount of 5G subscriptions due to the support of massive IoT deployments in 5G.

The findings of this study provide insights for the development of more robust NWDAF-supported mobility prediction ML models that can effectively defend against such attacks.

Acknowledgments

This work is supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research (SSF). We would like to thank Michael Liljenstam and Raaghul Ranganathan for their insightful feedback on the initial version of this paper.

References

[3GP19] 3GPP. *Study on the evolution of Cellular Internet of Things (CIoT) security for the 5G System*. Tech. rep. TR 33.861. 3rd Generation Partnership Project (3GPP), 2019.

[3GP22a] 3GPP. *5G System; Network Data Analytics Services; Stage 3*. Technical Specification (TS) 29.520. Version 17.8.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[3GP22b] 3GPP. *5G System; Study on enhancement for data collection for NR and ENDC; Stage 3*. Technical Specification (TS) 37.817. Version 17.0.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[3GP22c] 3GPP. *Architecture enhancements for 5G System (5GS) to support network data analytics services*. Technical Specification (TS) 23.288. Version 17.6.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[Air16] Airtel. *Airtel's Open Network*. 2016.

[Ant+21] E. Anthi et al. "Adversarial attacks on machine learning cybersecurity defences in industrial control systems." In: *Journal of Information Security and Applications* 58 (2021), p. 102717.

[Ash+22] I. Ashraf et al. "Zero Touch Networks to Realize Virtualization: Opportunities, Challenges, and Future Prospects." In: *IEEE Network* 36.6 (2022), pp. 251–259.

[Ati23a] S. A. Atiiq. *Dataset of Human Mobility and The Attack on NWDAF*. 2023.

[AWS06] J. Ariyakhajorn, P. Wannawilai, and C. Sathitwiriyawong. "A Comparative Study of Random Waypoint and Gauss-Markov Mobility Models in the Performance Evaluation of MANET." In: *2006 International Symposium on Communications and Information Technologies*. 2006, pp. 894–899.

[BFD20] M. Brisfors, S. Forsmark, and E. Dubrova. "How Deep Learning Helps Compromising USIM." In: *Smart Card Research and Advanced Applications: 19th International Conference, CARDIS 2020, Virtual Event, November 18–19, 2020, Revised Selected Papers*. 2020, pp. 135–150.

[Bre01] L. Breiman. "Random forests." In: *Machine learning* 45 (2001), pp. 5–32.

[CMS22] A. Chouman, D. M. Manias, and A. Shami. *Towards Supporting Intelligence in 5G/6G Core Networks: NWDAF Implementation and Initial Analysis*. 2022.

[CW17] N. Carlini and D. Wagner. "Towards Evaluating the Robustness of Neural Networks." In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 39–57.

[Deh+20] M. Dehghani Soltani et al. "An Orientation-Based Random Waypoint Model for User Mobility in Wireless Networks." In: *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2020, pp. 1–6.

[Ekm+08] F. Ekman et al. "Working Day Movement Model." In: *Proceedings of the 1st ACM SIGMOBILE Workshop on Mobility Models*. Hong Kong, Hong Kong, China, 2008, pp. 33–40.

[Eri20] Ericsson GAIA. *A framework for simulating mobility patterns in LTE*. Internal Link. 2020.

[Eur23] European Commission. *Principles of the GDPR*. 2023.

[Ge+16] X. Ge et al. "User Mobility Evaluation for 5G Small Cell Networks Based on Individual Mobility Model." In: *IEEE Journal on Selected Areas in Communications* 34.3 (2016), pp. 528–541.

[GEW06] P. Geurts, D. Ernst, and L. Wehenkel. "Extremely randomized trees." In: *Machine learning* 63 (2006), pp. 3–42.

[Gro+17] K. Grosse et al. "Adversarial examples for malware detection." In: *ESORICS: 22nd European Symposium on Research in Computer Security, 2017*. Springer. 2017, pp. 62–79.

[HV07] E. Hyytiä and J. Virtamo. "Random waypoint mobility model in cellular networks." In: *Wireless Networks* 13.2 (2007), pp. 177–188.

[HW79] J. A. Hartigan and M. A. Wong. "Algorithm AS 136: A k-means clustering algorithm." In: *Journal of the royal statistical society. series c (applied statistics)* 28.1 (1979), pp. 100–108.

[Jeo+21a] J. Jeong et al. "Mobility Prediction for 5G Core Networks." In: *IEEE Communications Standards Magazine* 5.1 (2021).

[Jeo+21b] J. Jeong et al. "Mobility prediction for 5g core networks." In: *IEEE Communications Standards Magazine* 5.1 (2021).

[Ke+17] G. Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[Kim+22] T. Kim et al. "An Implementation Study of Network Data Analytic Function in 5G." In: *IEEE International Conference on Consumer Electronics (ICCE)*. 2022.

[Kis+10] S. Kisilevich et al. "Spatio-temporal clustering." In: *Data Mining and Knowledge Discovery Handbook*. Ed. by O. Maimon and L. Rokach. Boston, MA: Springer US, 2010, pp. 855–874.

[Liu+15a] J. Liu et al. "Cloning 3G/4G sim cards with a pc and an oscilloscope: lessons learned in physical security." In: *BlackHat USA* (2015).

[Liu+15b] J. Liu et al. "Small Tweaks Do Not Help: Differential Power Analysis of MILENAGE Implementations in 3G/4G USIM Cards." In: *Computer Security – ESORICS 2015*. Cham: Springer International Publishing, 2015, pp. 468–480.

[LTZ08] F. T. Liu, K. M. Ting, and Z.-H. Zhou. "Isolation Forest." In: *2008 Eighth IEEE International Conference on Data Mining*. 2008, pp. 413–422.

[Lu+21b] Y. Lu et al. "A K-means Clustering Optimization Algorithm for Spatiotemporal Trajectory Data." In: *Human Centered Computing*. Cham: Springer International Publishing, 2021, pp. 103–113.

[MTB23] A. Mensi, D. M. Tax, and M. Bicego. "Detecting outliers from pairwise proximities: Proximity isolation forests." In: *Pattern Recognition* 138 (2023), p. 109334.

[Nai+05] P. Nain et al. "Properties of random direction models." In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 3. 2005.

[P1 21] P1 Labs Reserach. *Mobile authentication vectors distribution across roaming interfaces*. Tech. rep. B02. P1 Labs Reserach, 2021.

[Pap+16] N. Papernot et al. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks." In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 582–597.

[Sha+20] R. Shafin et al. "Artificial intelligence-enabled cellular networks: A critical path to beyond-5G and 6G." In: *IEEE Wireless Communications* (2020).

[Sou+21] W. Soussi et al. "Moving target defense as a proactive defense element for beyond 5G." In: *IEEE Communications Standards Magazine* 5.3 (2021), pp. 72–79.

[ST19a] G. Solmaz and D. Turgut. "A Survey of Human Mobility Models." In: *IEEE Access* 7 (2019), pp. 125711–125731.

[STL20] J. Sternby, E. Thormarker, and M. Liljenstam. "Anomaly detection forest." In: *ECAI 2020*. IOS Press, 2020, pp. 1507–1514.

[Sze+14] C. Szegedy et al. "Intriguing properties of neural networks." In: *International Conference on Learning Representations*. 2014.

[Wan+21a] C. Wang et al. “FLAML: A Fast and Lightweight AutoML Library.” In: *MLSys*. 2021.

[Wec20] S. Weckert. *Google Maps Hacks*. 2020.

[Zha+21] J. Zhao et al. “SecureSIM: Rethinking Authentication and Access Control for SIM/ESIM.” In: *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. MobiCom ’21. New Orleans, Louisiana, 2021, pp. 451–464.

[Zol+23] M. Zolotukhin et al. “On Attacking Future 5G Networks with Adversarial Examples: Survey.” In: *Network* 3.1 (2023), pp. 39–90.

AutoML in the Face of Adversity: Securing Mobility Predictions in NWDAF

Abstract

Network Data Analytics Function (NWDAF) is a key component in 5G networks, introduced by 3G Partnership Project (3GPP) standards, that leverages machine learning to optimize network performance. The 3GPP standards mandate that mobile network operators should retrain NWDAF models to maintain accuracy. However, the presence of adversarial user equipment (UE) can introduce corrupted data points during this retraining process, compromising prediction accuracy. Manually selecting optimal models for NWDAF tasks is challenging and time-consuming, making Automated Machine Learning (AutoML) an attractive solution. This paper investigates strategies for operators to maintain NWDAF model performance using AutoML in the face of adversarial attacks, focusing on mobility prediction. We consider two main retraining strategies: (A) reselecting the model using AutoML at each retraining and (B) retaining the initial model. We evaluate these strategies using different AutoML frameworks under varying proportions of adversarial UEs, assuming that the attacker is unaware of the retraining schedule and the operator lacks the capability to distinguish between adversarial and legitimate mobility patterns. Our results show that, in the presence of adversarial UEs, retraining with AutoML yields worse results compared to retaining a well-trained initial model selected using an extensive AutoML search during ini-

Syafiq Al Atiiq, Christian Gehrmann, Yachao Yuan, Jakob Sternby. "AutoML in the Face of Adversity: Securing Mobility Predictions in NWDAF". In 9^{th} *International Conference on Fog and Mobile Edge Computing, FMEC 2024, Malmö, Sweden*. pp. 90-98, IEEE.

tial training. We, therefore, recommend that operators prioritize model selection during initial training, ensuring the base model is optimally tuned to maintain accuracy over subsequent retraining. This allows effective mobility prediction to be preserved even if corrupted data cannot be fully excluded.

1 Introduction

5G network has brought significant advancements in mobile technology, with the 3GPP introducing the NWDAF [3GP22c; 3GP22a] as a new component. NWDAF harnesses the power of machine learning (ML) to optimize network performance, enabling operators to enhance efficiency, reduce costs, and improve user experience. However, the 3GPP standards require operators to periodically retrain NWDAF models to ensure continued accuracy, a task that can be complex and resource-intensive when done manually. AutoML has emerged as a promising solution, with recent reports [CI22; Ora22] highlighting its growing adoption in NWDAF deployments. One of the key applications of NWDAF is mobility trajectory prediction, which involves forecasting the future locations of user equipment (UE). Accurate mobility predictions are essential for optimizing network operations and resource allocation. For instance, by leveraging mobility predictions, operators can achieve significant reductions in paging signaling, leading to improved computational efficiency [Man+23].

However, the presence of malicious UEs can compromise the integrity of the NWDAF retraining process by injecting corrupted data points. While previous research [Ati+23] has demonstrated the detrimental impact of adversarial mobility on prediction accuracy during the inference phase, this paper takes a step further by investigating the effects of adversarial attacks on the training data itself. We focus on scenarios where malicious UEs evade detection until the retraining phase, corrupting the data used to update the NWDAF models. In our study, we assume that the attacker is unaware of the operator's retraining schedule and that the operator lacks the capability to distinguish between adversarial and legitimate mobility patterns.

The 3GPP standards recognize the potential threat posed by abnormal UEs and mandate that operators implement mechanisms to identify and exclude their data to maintain the accuracy of NWDAF analytics (3GPP 23.288 [3GP22c], section 5.1). However, given the challenges associated with anomaly detection, even the presence of such mechanisms may not guarantee the identification of all malicious activities within the network. This paper delves into the challenge of maintaining NWDAF model performance in the face of adversarial attacks that evade detection, focusing on the role of AutoML in discovering robust models.

We explore two primary retraining strategies for operators to mitigate the impact of adversarial attacks: (A) employing AutoML to reselect the model at each retraining iteration and (B) retaining the initial model discovered by AutoML. We assess the effectiveness of these strategies using various AutoML frameworks and

under different levels of adversarial UE presence. Our findings reveal that, in the presence of adversarial UEs, retraining with AutoML yields worse results compared to retaining a well-trained initial model selected using an extensive AutoML search during initial training.

The main contributions of this paper are twofold:

- We conduct an in-depth investigation into the impact of adversarial UEs on the AutoML model selection process for NWDAF mobility prediction, considering scenarios where operators rely on AutoML for model discovery and lack the capability to distinguish between adversarial and legitimate mobility patterns.
- Based on our analysis, we provide recommendations on the most effective strategy for operators to maintain NWDAF model accuracy in the presence of adversarial UEs, focusing on the trade-offs between retraining with AutoML and retaining a well-trained initial model.

The remainder of this paper is structured as follows. Section 2 surveys related work on AutoML, mobility prediction, and adversarial machine learning. Section 3 provides an overview of the NWDAF model lifecycle and the problem statement. Section 4 describes the operator retraining strategies considered in this study. Section 5 presents the threat model. Section 6 details the modeling of UE mobility. Section 7 discusses the simulation setup and results. Finally, Section 8 concludes the paper and outlines future research directions.

2 Related Work

Below, we discuss the most relevant related work. We will cover previous work on AutoML with a particular focus on open frameworks. Next, we discuss previous work on ML-based mobility prediction and, finally, adversarial machine learning.

2.1 Automated Machine Learning (AutoML)

AutoML [SJ22] has gained significant attention in recent years due to the increasing demand for ML solutions and the shortage of experts [Tim21]. It is designed to automate the process of building machine learning models, including tasks such as data preprocessing, feature engineering, model selection, and hyperparameter tuning [HZC21; Kar+21]. Numerous approaches have been proposed in the literature to achieve this goal, such as genetic algorithms [MM19], Bayesian optimization [Sha+15a], and cost frugal optimization [WWH21]. It is particularly well suited for telecom operators, especially small ones, where AI is not the focus, and acquiring such expertise might take time, effort, and money. While previous research has provided a benchmark on different AutoML frameworks [Gij+22],

the result was from general datasets and runs on a malignant environment. Our work focuses on exploring AutoML for mobility data in mobile operators while adversarial data is introduced to the environment simultaneously.

Framework	Optimization and Search Space
Auto-sklearn [Feu+22]	Bayesian Optimization
Autogluon [Eri+20]	Stacked Ensembles of Preset Pipelines
FLAML [Wan+21b]	Cost Frugal Optimization

Table 1: Employed AutoML frameworks in the experiments.

Given the excessive costs involved in assessing every framework, we chose to focus on analyzing three specific frameworks for this study from table 1, i.e., Auto-sklearn [Feu+22], Autogluon [Eri+20], and FLAML [Wan+21b]. We focus on finding a generalized conclusion on how the AutoML solution would handle when the training data is poisoned with adversarial data points, particularly in the context of NWDAF in 5G.

Each of the frameworks listed above employs unique strategies: Auto-sklearn employs Bayesian optimization [Sha+15a], and AutoGluon [Eri+20] employs multiple techniques, i.e., ensembling and stacking [Pav18] for model performance enhancement, alongside multi-fidelity optimization [LL24] and bandit-based resource allocation [Cha+08] for efficient model selection. It also utilizes Neural Architecture Search (NAS) [EMH19] for optimizing models to balance performance and search efficiency. FLAML is a lightweight AutoML tool that employs an adaptive ensemble learning approach to construct high-performing models with minimal computational overhead [WWH21].

2.2 ML-based mobility prediction

In the area of ML-based mobility prediction and its resource application in 5G, several works have been proposed. The existing works can be roughly categorized into two groups: traditional ML-based [Imt+18; KK10; Gho+14], and deep learning-based [AK20; Ozt+19]. An example of the former is a new framework for 5G using Random Forest to improve its resource allocation efficiency and robustness [Imt+18]. Additionally, [Gho+14] introduces an approach for predicting node mobility in mobile ad hoc networks using Extreme Learning Machines (ELMs) [HZS06]. For the latter approaches, in [AK20], Long Short-Term Memory (LSTM) was employed for traffic prediction in vehicular ad hoc networks (VANETs). The authors of [Ozt+19] proposed a low-cost approach to mobility prediction in 5G using a deep learning model based on the user's previous location and velocity data.

2.3 Adversarial Machine Learning

Relative to the stages of learning, a report from the National Institute of Standards and Technology (NIST) [OV23] classifies the attack toward machine learning model into two categories: (i) Training-time attacks and (ii) Deployment-time attacks. The common term of the former is poisoning attack [BNL12], which can be classified into two categories: data poisoning [BNL12; Gu+19] where an attacker injects harmful data into the training set and model poisoning [Liu+18a] where the attacker tampers with the model itself, either during training or via a compromised update mechanism. The latter can also be classified into two categories: evasion attack [Big+13; GSS14; Sze+13], where the adversary modifies input data to cause the machine learning model to make incorrect predictions, and privacy attack [Sho+17; DN03] where the aim is to extract sensitive information from the model or its data. While the work against a deployed NWDAF model [Ati+23] is considered to be an evasion attack, our work in this paper focuses on data poisoning attacks.

3 Background and Problem Definition

3.1 Background: The Lifecycle of an NWDAF Model

The NWDAF model is defined by 3GPP [3GP22c]. It is a rather broad framework allowing different mobile network analytics functions to be implemented. The standard also discusses a mechanism for identifying abnormal UEs. There is a mandatory requirement for the operator to have the means to distinguish such abnormality within a given time window and potentially exclude their data to maintain the accuracy of NWDAF analytics (3GPP 23.288 [3GP22c], section 5.1). As anomaly detection is challenging, even if such a mechanism exists, it cannot detect all malicious activities in the network.

The life cycle of an NWDAF model following the standard [3GP22c] would typically look like the one we depict in Figure 1. Below, we explain the details of the different steps:

C1 The cycle starts with an initial training (green) to generate the first version of the trajectory model. The model selection can be handcrafted manually or by using an automated mechanism with AutoML. In this paper, our approach follows the model selection of the prior work [Ati+23], which involves picking the best model with AutoML. Once the best model is identified, it is deployed to the live environment to provide services to other NFs in the network. The accuracy of the deployed model is periodically checked and remeasured every predefined period. A decrease in accuracy indicates either behavioral changes of legitimate UEs or the presence of adversarial UEs in the network.

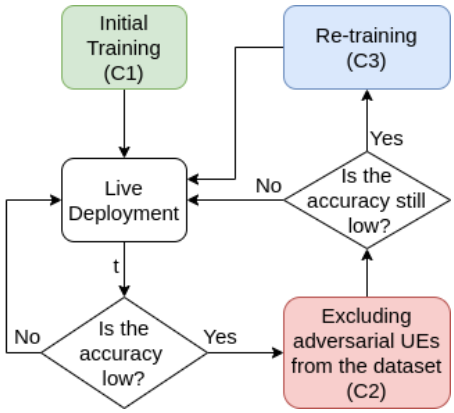

Figure 1: The lifecycle of NWDAF's model

C2 In the presence of adversarial UEs in the network, the 3GPP standard mandates that the operator must have the means to distinguish the legitimate and adversarial UEs to be potentially excluded from the network (red).

C3 If the accuracy is back to normal after the exclusion of adversarial UEs (C2), the NWDAF model continues its operation. Otherwise, the operator assumes that there is a shift in the (legitimate) user mobility behavior. When this happens, the assumed-to-be all benign UEs form the training set and will be used in the next retraining task (blue). This generates a new model that will be deployed to the live environment. Similar to the C1 situation, in this phase, the model selection can be handcrafted manually, or AutoML can be used to automate the selection.

3.2 Problem Definition

In this paper, we assume the situation in step C2 where the operator is unable to exclude abnormal UEs from the dataset. This means the adversarial data points are included in the next retraining task in C3. This situation can, for instance, occur due to (i) a newer version of adversarial mobility is introduced to the network such that the defense mechanism fails to recognize, or (ii) a number of legitimate UEs turns into adversarial in the middle of NWDAF deployment in a way that the system believes that the UEs are still benign. When no proper anomaly detection exists, the NWDAF will train on all data, including any adversarial data points, and as a consequence, the adversarial data points will poison the dataset for the next retraining task in C3. The severity of this issue depends on (i) the type of adversarial mobility (previous work [Ati+23] has shown that different mobility affects the accuracy during inference in different ways) and (ii) the proportion of the adversarial data points in the dataset.

Given the data poisoning situation described above, we consider the problem of forming an operator retraining strategy in C3 that gives acceptable mobility prediction accuracy. The term acceptable here means that the accuracy for the legitimate user should not be decreased in any way. Even when it is, the decrement should be kept as small as possible to minimize the impact. In the context of the use of AutoML, we are particularly interested in whether the use of AutoML in C1 and C3 is suitable when the operator is enduring an attack from the adversarial UEs. This includes whether they need to employ AutoML at all times (both C1 and C3) or not. Using AutoML at all times is convenient, but we want to find the answer to whether using this strategy is robust enough in the long run, particularly when the number of adversarial data points increases. It is also important to notice that *if*, we can find a solution to this problem; there is not even a *need* to introduce any other type of anomaly detection into the system for this type of mobility attack, as the system will be robust *independent* of the presence of the attack or not; as long as the selected model is robust enough.

4 Different Operator Retraining Strategies

To address the identified research problem, we investigate two natural classes and three different types of hypothetical operator strategies based on the characteristics of their retraining profile in C3, i.e., how they use AutoML during retraining tasks. We denote the two classes, A and B, respectively, and the selection is based on a more general work about continual learning in practice [Die+18]. But, before that, we explain the time budget in AutoML as it is the main factor for classifying the strategies.

4.1 Time-budget in AutoML

The term *time-budget* [Feu+18] refers to the maximum amount of time that the AutoML system is allowed to search for the best machine learning model and its hyperparameters given the input dataset. Essentially, it sets a runtime limit on the AutoML process. However, searching through all possible combinations of models and their hyperparameters can be computationally expensive and time-consuming. While a more extended time budget can potentially lead to a more optimized model, it does not always guarantee significantly better performance. After a certain point, the improvements might be marginal. Hence, choosing an appropriate time budget is crucial to balance model performance and computational efficiency.

4.2 New model and data for each retraining

The first strategy, A, is about constantly updating the learning process as new data comes in. In [Die+18], it is termed as a self-correction policy where "*the policy*

engine is responsible for updating models by triggering retraining or other actions". In our case, the trigger is a lowered accuracy after the system is assumed to have separated the adversarial data points from the dataset (which is actually not the case). With the model being updated for every trigger of the policy engine, AutoML is used to select the best model at all times (both during initial training and at retraining). Strategy A is formally defined as:

A The operator selects the best model and hyperparameters both during initial training (C1) and during the retraining period (C3) using AutoML.

This is a natural and straightforward strategy to choose. In this case, setting an appropriate AutoML time budget is critical. Performing a retraining task using AutoML can be time-consuming, with a significant portion of time dedicated to finding the best model given the polluted dataset. In order to minimize the impact, a lower time budget is desirable. However, a lower time budget may result in a less accurate model and a poor selection of hyperparameters. Striking a balance between these two factors is crucial to achieving a performant model while keeping uptime high. The discussion to balance these two factors is explored in section 7.3.

4.3 Old model and new data for each retraining

The second strategy, B, is the opposite of the first one. It diverges from the continual learning ideal by limiting the scope of retraining. This reflects a practical compromise, as highlighted in [Die+18], between the ideal of continuous model evolution and the realities of computational and resource constraints. According to the second strategy, the operator thinks model reselection using AutoML is costly (computation, time, and cost-wise). In this case, the operator's natural choice is to select the initial model with AutoML but retrain using the same selected model as picked initially with newer data (which, in this case, might include the adversarial data points). We formally define strategy B as follows:

B The operator selects the best model and hyperparameters with AutoML only during initial training (C1) and then retrains the same initial model and hyperparameters during the retraining period (C3) with newer data.

We divide this retraining strategy class into two different categories, depending on the allocated time budget for the initial training (C1):

B1 Short time budget.

B2 Long time budget.

For the strategies B1 and B2, as the operator decided to stick with the initial model, allocating more time budget is not a problem. This is because AutoML

is only used when NWDAF is not yet deployed, and spending as much time as possible in finding a suitable model can be done without affecting the live deployment in any way. Hence, allocating a larger time budget is not as costly as in the first strategy. The main consideration is, instead, the computational cost, where a larger time budget means more resources are needed for the initial training.

5 Threat Model

For a mobility attack to be considered feasible, it must not require an unreasonable amount of effort. In particular, an attack that involves moving UEs (physically) throughout the network for extended durations is not practical. The resources and effort needed to carry out such an attack would outweigh any potential gains, making these types of attacks improbable and beyond the scope of this analysis.

This study operates under the assumption that an adversary possesses the capability to obtain UEs and create multiple clones with identical International Mobile Subscriber Identity (IMSI) numbers. The attack can be scaled vertically by acquiring more UEs or horizontally by increasing the number of cloned UEs. The acquisition process can be either legitimate, following standard procedures, or illegitimate if existing ones are stolen. Note that this scaling is limited as assuming the adversary can control all the UEs in the network is not feasible. It means that the adversary does not have full white-box capabilities, as it only controls a small portion of the UEs in the network. Research has demonstrated the feasibility of extracting the main secret key from a 4G Universal Subscriber Identity Module (USIM) [Liu+15a; Liu+15b]. If the adversary gains access to the primary USIM key, they may be able to duplicate it. While NWDAF and our attack are mainly intended for 5G, we envision that a similar attack can be applied in a non-standalone 5G network.

Since our primary objective is to illustrate how the presence of adversarial UEs can also impact the accuracy of legitimate UEs' trajectory prediction **if** the operator fails to exclude the adversarial UEs before the next retraining task, we assume that the adversarial mobility from the attacker cannot be detected in step C2, i.e., a data poisoning event has occurred. In such a worst-case scenario (given the attack that is performed), we explore what can be done to minimize the impact on the operator using the different identified strategies A, B1, and B2. In a real-world scenario, this situation often occurs when a newly formed adversarial pattern is introduced by the adversary, making the system unable to detect the attack. It should be noted that the mobility model itself is not a part of the NWDAF specification, as the 3GPP standard only defines interfaces and use cases. However, we have defined a mobility model with NWDAF input and output.

6 Mobility

6.1 Mobility Simulator

The privacy-sensitive nature of UE mobility data poses challenges to obtaining real-world data from mobile operators due to strict privacy laws, such as the General Data Protection Regulation (GDPR) [Eur23]. To address this issue, Ericsson has developed an internal spatiotemporal mobility simulator [Eri20] capable of producing UE mobility data based on predetermined mobility models. The spatial aspect of the simulator is based on a real-world deployment of Airtel's open-network topology [Air16] in India. On top of that, the mobility data is generated as time series data that manifests UE connected to a particular eNodeB with calculated signal strength based on distance and the load of the respective eNodeB. Although the simulator is not open-sourced, a subset of the post-processed dataset is available in our repository: `https://github.com/nwdaf-research/dataset-attack`.

6.2 Prediction Accuracy

Mobility prediction in UE involves two crucial metrics: (1) location prediction at a specific time denoted as $\hat{l}$, and (2) timeslot prediction ($\hat{s}$) given that $\hat{l}$ is accurate. The latter metric indicates how long a UE remains connected to a specific eNodeB at a particular time. For a given period of $t_2 - t_1 \in T$ with n mobility events e, the mobility prediction accuracy is calculated as the sum of correct timeslot predictions $\hat{s}$ (when $\hat{l}$ is correct) divided by the total number of predictions. The accuracy is shown in equation 1 and will be used throughout this paper.

$$\textbf{Accuracy} = \frac{\sum_{e=0}^{n}(\hat{s}_{correct}|\hat{l}_{correct})}{n} \tag{1}$$

6.3 Legitimate Mobility

Generalizing UE mobility and providing a one-size-fits-all model to perform prediction is a non-trivial task [ST19b]. This is due to the uniqueness of the mobility, either as an individual or as aggregates in a particular area. While some mobilities are too stochastic to enable reliable prediction, others may follow patterns that are possible to model. With this assumption, we decided to pick a subset of UE mobility called *working professional* [Ekm+08] as the legitimate UEs in the dataset, which follows a periodic movement pattern between the office and the home. It assumes that professionals engage in regular movements between their residences and workplaces throughout the day and evening. The model employs randomly selected initial coordinates for home and office, adhering to specific location constraints. This mobility pattern is relevant in most areas worldwide. As our base

installation of the eNodeB is based on an operator in India, this mobility represents the population of 140 million people based on Census data [Ruk16].

6.4 Adversarial Mobility

For adversarial mobility, the previous study [Ati+23] has shown one attack, called the tuple jump mobility attack, as the most effective in decreasing the NWDAF's accuracy. We use this type of attack in our measurement. To do this, the attacker divides the adversarial UEs into pairs and assigns each pair the same IMSI (original and cloned), allowing them to impersonate each other. Within each pair, the adversary can make one device connect to an eNodeB while the other remains silent, then switch their roles in the next step, creating the appearance of movement between different locations while actually both of them stay in their locations at all times. The attacker makes the choice of eNodeBs randomly and can include geographically distant ones. This attack can be carried out without much effort since the devices can be placed near the chosen eNodeBs and do not need to move physically, fulfilling the requirement from the adversary in our threat model. When an adversary obtains multiple UEs, they can use this technique to confuse the network operator and make it seem like a single device is moving erratically between different eNodeBs.

7 Simulation and Result

7.1 Simulation Environment

The UE mobility simulation and the computation to find the best model, followed by attacking such a model, was conducted on a server with 48 cores of AMD EPYC 7413 and 256 GB of memory. While the number of adversarial UEs varied between 50 and 1000 (with an increment of 50), the simulation included a baseline of 10,000 legitimate UEs. The simulation spanned five days, with the first four days used for training and the final day for testing.

7.2 Data Preprocessing

This work utilized a dataset sourced from the mobility simulator mentioned in section 6.1, containing three main features: {IMSI, eNodeB_path, signal_strength}. The data contains historically connected eNodeBs coupled with each connection's timestamp and signal strength for a particular unique IMSI. Table 2 shows an IMSI 08126630, which is historically connected to eNodeB 2 and 17 at timestamp 99 and 244, respectively, where the signal strength of each connection is 0.154 and 0.199.

Given many available IMSIs, the dataset is then transformed from an IMSI-based to an event-based. The reason is that we want to predict the future location of a particular IMSI at a particular time. In this case, it is better to have the

IMSI	eNodeB_path	signal_strength
08126630	'99':2,'244':17,...	'99':0.154,'244':0.199,...

Table 2: Raw data

timestamp (event-based) as the main identifier of each entry of the data instead of the IMSI. So, for a particular timestamp, the transformed dataset contains many IMSIs connected to many eNodeBs, but one IMSI is only allowed to connect to one eNodeB. The duration an IMSI remains connected to one eNodeB before switching to another is called a timeslot, as used in equation 1.

To provide a better context for each event, we then enrich the dataset with additional features, i.e.:

- `enode_1-enode_4`: The history of 4 previous connected eNodeBs.
- `time_1-time_4`: The timeslot of each connection.
- `sig_st`: Signal strength of that particular connection to the target eNodeB.
- `imsi`: The IMSI of the connected UE of the target eNodeB.
- `neigh_1-neigh_4`: Top four neighboring eNodeBs of the currently connected eNodeB.
- `early_morning`, `morning`, `noon`, `evening`, and `night`: Categorizing each timestamp into five different bins.
- `home_enb`: A home eNodeB where a particular IMSI stays connected the longest.

The enriched features are included in the final input variable, represented as x: {`enode_1, enode_2, enode_3, enode_4, time_1, time_2, time_3, time_4, sig_st, imsi, home_enb, early_morn, morning, noon, evening, night, neigh_1, neigh_2, neigh_3, neigh_4`}

The dataset is subsequently employed to construct a model that predicts the binned timeslot ($\hat{s}$), during which a UE maintains a connection to a specific eNodeB ($\hat{l}$). Note that the timeslot is binned to reduce the effect of small differences in continuous values, which are of less importance. Also, as the timeslot is the time spent on each eNodeB before moving to another location, classifying binned values is simpler than classifying the continuous values of many different UEs. This comes at the cost of rounding up each value to the respective range in the binned position. Note that a round-down is possible, but the resource will be allocated less than needed.

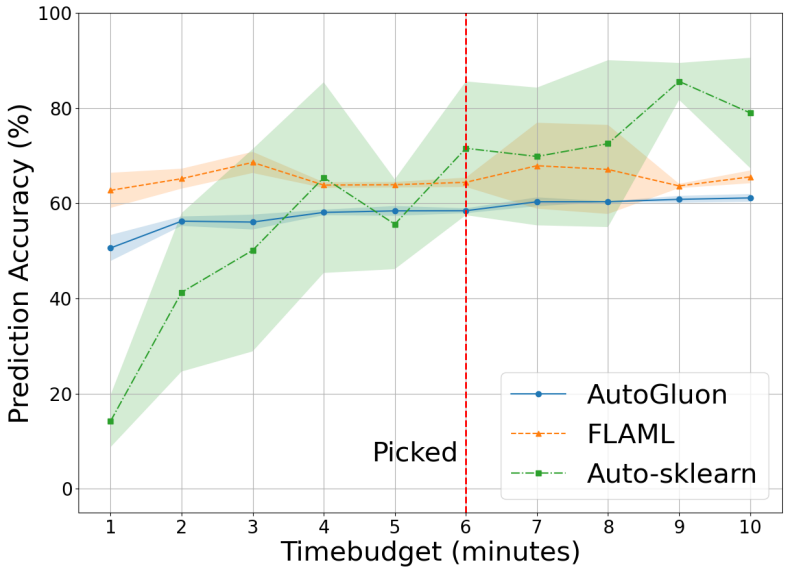

Figure 2: Time budget to the accuracy

7.3 Finding the optimum time budget

Given the discussion in section 4, setting a suitable time budget for the model selection is important. In particular, it is more important for the retraining task in C3 because it is triggered when the accuracy of the prediction is low, while at the same time, the NWDAF is serving the subscribed NFs. On the other hand, for the initial model selection in C1, one would be able to spend as much time as possible due to the model not yet being operational. In this section, we discuss how we set the time budget for all the strategies A, B1, and B2 in relation to the initial and retraining task.

To start with, we explore how different time budgets impact prediction accuracy when the training set is not poisoned, i.e., consisting exclusively of legitimate UEs. We vary the time budget from 1 to 10 minutes and observe the changes in accuracy over the three different AutoML frameworks mentioned in section 2.1: Auto-sklearn, FLAML, and AutoGluon. Resource-wise, it is not feasible to test a very large number of different frameworks. We have chosen to work with these three as they give a good representation of different AutoML algorithms and use fundamental different optimizations and search spaces as shown in table 1.

Then, we pick the smallest time budget once the values start to converge among the three frameworks. It would be possible to pick a longer time, but a shorter time budget is desired (as the time taken to iterate the process in Figure 1

is also shorter), especially during the retraining period in C3.

Figure 2 illustrates this relationship, showing a marked increase in accuracy in the initial runs, with a convergence over all the frameworks around the 6-minute mark. After this, the accuracy among the frameworks does not have a significant difference. To this end, we standardized the minimum time budget to get a meaningful result and obtain a performant model: 6 minutes. This is applied to both strategies A and B1 (for simplicity, let's call them TA and TB1, respectively). In A, TA is used in both C1 and C3, but in B1, TB1 is used only in C1 as the retraining in B1 uses the same selected model picked initially. Note that this is the minimum amount of time to be spent in model selection using AutoML to select the best-performing model. In these experiments, however, different datasets might yield different minimum time budget requirements. Adversary knowing the information of applied time budget to the training strategy could trigger the retraining phase more frequently, resulting in a never-ending cycle of retraining, which might affect the NWDAF's performance. Such a case is out of the scope of this paper and left for future investigation.

For a longer time budget in B2, we heuristically set it to be 10x the minimum time previously mentioned. Thus, the time budget for B2 is 60 minutes; we call it TB2 (used only in C1). Note that there is no long time budget for strategy A because spending longer time during C3 in strategy A is undesired, especially when the accuracy of the prediction is, at the same time, low. Using TA, TB1, and TB2, we explore the operator's options for preserving the NWDAF's model prediction accuracy to the best possible extent. Our ultimate goal is to provide recommendations on which strategy to pursue for mobile operators facing similar issues in the future.

7.4 Attack on the training set

Given the pre-processed data and the minimum time budget mentioned earlier, we investigate the impact of the attacks on the hypothetical operators with the strategies A, B1, and B2. As the operator is assumed to be unable to distinguish the adversarial UEs, we investigate the effect of data poisoning during retraining. The poisoning is performed with an increasing proportion of adversarial UEs to the total number of legitimate UEs in the network, i.e., as there are 10000 legitimate UEs and 50 to 1000 adversarial UEs, the attack is performed with the proportion of 0.005 to 0.1 adversarial UEs to the total legitimate UEs. The interval of 50 is selected to provide a granular understanding of how the NWDAF's performance is affected as the number of adversarial UEs increases within this range. To make a fair comparison, the configuration is kept to be as basic as possible for each framework we run. The following is the detail of the run on each strategy:

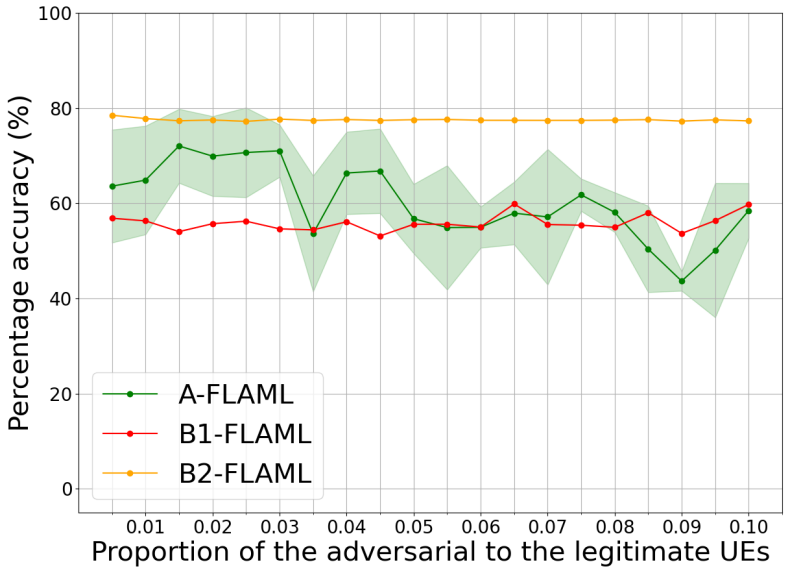

Figure 3: The proportion of adversarial UEs on the accuracy of the retrained model using FLAML AutoML framework.

A

For each proportion of adversarial UEs, the retraining task with AutoML is performed five times, and then the mean and the standard deviation of the accuracy are calculated. Since AutoML is involved, each run for this strategy yields a different model and hyperparameters.

B1 and B2

For each proportion of adversarial UEs, the retraining task is performed five times with the same model selection as selected by the initial training, i.e., the same model but with different training data. The initial training for the model selection still involves AutoML with a time budget of 6 minutes for B1 and 60 minutes for B2. In the end, the mean and the standard deviation of the accuracy for each proportion of adversarial UEs are calculated.

7.5 Results

Given the attack previously explained, we draw the line plots of the prediction accuracy relative to the proportion of adversarial UEs in the network. The line plots in figures 3, 4, and 5 display two key pieces of information. The X-axis shows

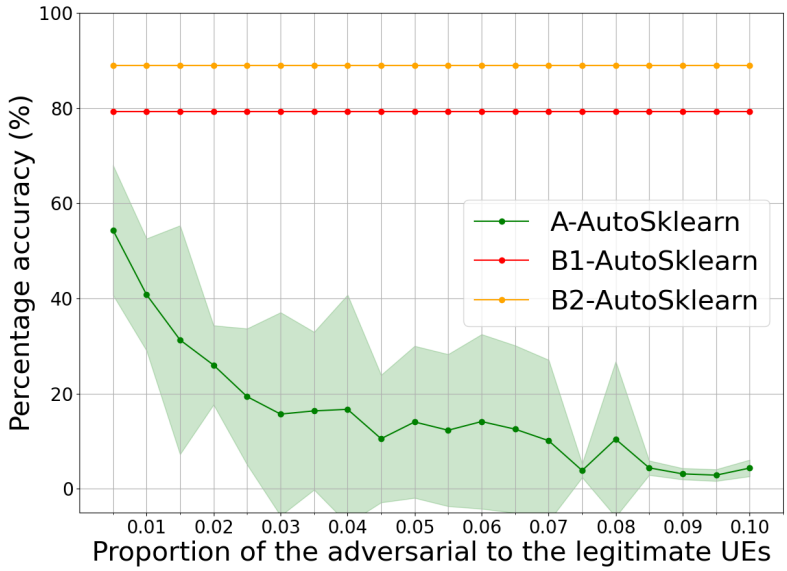

Figure 4: The proportion of adversarial UEs on the accuracy of the retrained model using Auto-sklearn AutoML framework.

the ratio of adversarial UEs to the total legitimate UEs available in the network. On the Y-axis, the accuracy corresponds to each of these ratios.

Given the 6 minutes time budget explained in 7.3, it is clear that retraining the poisoned data with the AutoML frameworks for each retraining occasion (scenario A) is not enough to keep the accuracy desirable and cannot be generalized as the solution. While FLAML shows a slight decrease over the x-axis, AutoSklearn shows a significant decrease as the proportion of adversarial UEs increases. On the other hand, AutoGluon shows a more stable accuracy even though the accuracy value is around 42% regardless of the proportion of the adversarial UEs. The strategy A for FLAML and AutoSklearn shows a considerable variance compared to the rest of the strategies. The variability of the accuracy could be attributed to the way the frameworks explore the model space, i.e., (i) Each run may start with a different random state, leading to different paths in the search for the optimal model, and (ii) The frameworks employ a cost-effective search strategy that balances between exploring different models and exploiting the best-performing models. This strategy can involve probabilistic decisions that might differ from one run to another. However, the accuracy is generally dropped (FLAML and AutoSklearn) or already low (AutoGluon) even when the model is newly selected for each proportion of adversarial UEs. In this case, following the self-correction

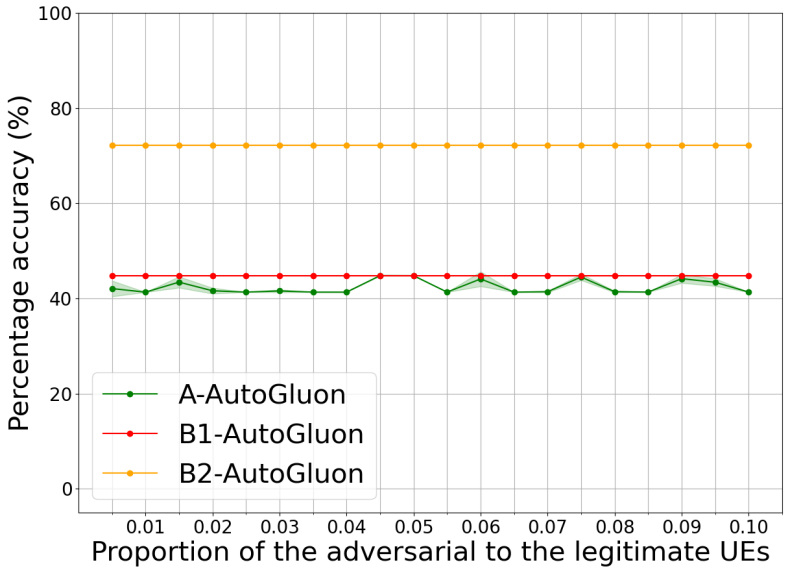

Figure 5: The proportion of adversarial UEs on the accuracy of the retrained model using AutoGluon AutoML framework.

policy [Die+18] does not suit the situation, particularly when the proportion of the adversarial UEs is high.

Maintaining the original model and retraining it with poisoned data helps to keep the accuracy stable. Specifically, in scenario B1, all evaluated frameworks manage to maintain their accuracy levels, even as the share of adversarial UEs increases. However, FLAML and AutoSklearn achieve higher accuracy compared to AutoGluon under these conditions. AutoGluon, in contrast, performs poorly, with its accuracy being worse than random guessing from the start. This reveals two key insights: firstly, having up to a 10% proportion of adversarial UEs does not significantly affect model performance in B1, despite the training data being poisoned with an increasing number of adversarial UEs. Secondly, in this specific case, AutoSklearn and FLAML outperform AutoGluon in developing a more accurate model, even though all frameworks manage to maintain consistent accuracy as the proportion of adversarial UEs increases.

However, it is clear from all the figures that B2 is the most robust scenario among all the strategies. Of all the frameworks, not only does B2 demonstrate the highest accuracy in comparison to the other scenarios, but it also maintains this accuracy consistently, even when the training data is incrementally poisoned. This is attributable to (i) a more extensive search and (ii) better hyperparameter

tuning. The former allows AutoML to try out more models and a wider range of hyperparameters. This increases the likelihood of finding a model that generalizes better to unseen data, even when the data is poisoned with a larger proportion of adversarial UEs. The latter means AutoML can spend a longer time tuning the hyperparameters of each model. Hyperparameter tuning is critical for getting the best performance out of a machine learning model, and more time can lead to more precise tuning, resulting in a more robust model, even in the presence of adversarial data points.

While our results demonstrate that the B2 strategy, which involves an extensive initial AutoML search followed by retraining with the same model and hyperparameters on updated data, provides the best resilience against data poisoning attacks, it is important to consider the potential limitations. Although the B2 strategy updates the model with new data during retraining, it may not fully capture the context drift in the legitimate user data over time, as the model architecture and hyperparameters remain fixed. As the mobility patterns of legitimate users evolve, the fixed model might not be able to adapt optimally to these changes, potentially leading to a gradual decrease in prediction accuracy for those users.

To mitigate this issue, periodic retraining using AutoML on carefully filtered data could help discover new model architectures and hyperparameters that better adapt to evolving legitimate user patterns while still maintaining robustness against attacks. Additionally, employing unsupervised anomaly detection techniques to identify and isolate potential adversarial data points before retraining could allow the model to benefit from updates that capture context drift more effectively while minimizing the impact of data poisoning. Future research should explore methods to balance the need for model adaptability with robustness against data poisoning attacks in the context of NWDAF mobility prediction to provide mobile network operators with strategies that optimize both factors.

8 Conclusions

This paper investigates different AutoML frameworks for data poisoning of mobility data. When the adversarial UEs are not promptly identified and excluded from the training dataset, the quality of the mobility predictions for legitimate UEs deteriorates. This is particularly true if the trajectory model's retraining process is not robust enough. We show that while AutoML frameworks are effective in initial model selection, their efficiency during retraining is significantly challenged by the presence of adversarial mobility data. Extensive exploration of the model space and meticulous hyperparameter tuning at the beginning, as shown in operator B2, displayed a marked improvement in maintaining prediction accuracy, even when the model is fed with adversarial data points during the retraining phase.

Our study suggests that while automating model selection in the retraining phase is handy, the fact that the model owner has a limited time to train a newer model makes it problematic, especially when faced with adversarial data points.

For future mobile network operations, our research recommends that mobile operators optimize the use of AutoML only during the initial training phase when NWDAF has not been deployed. This is to ensure that the operator gets the best trajectory model without worrying about the time budget, as shown by B2. Hence, even when the operator is being attacked by adversarial UEs, poisoned training data does not harm the overall system performance.

Future works include exploring the same strategy using real-world datasets from an actual NWDAF deployment. This is needed to validate our conclusion in a real-world scenario. While we only explore Auto-sklearn, FLAML, and AutoGluon, we find that other frameworks likely show similar trends, but still, a scrutinization is needed to confirm our hypothesis. Also, with a larger resource, one might be able to find the correlation between how much time is required to keep the accuracy desirable if the operator wants to stick with the scenario A.

Acknowledgements

This work is supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research and the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101139067. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them. The computations and data handling were enabled by resources provided by LUNARC, The Centre for Scientific and Technical Computing at Lund University.

References

[3GP22a] 3GPP. *5G System; Network Data Analytics Services; Stage 3*. Technical Specification (TS) 29.520. Version 17.8.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[3GP22c] 3GPP. *Architecture enhancements for 5G System (5GS) to support network data analytics services*. Technical Specification (TS) 23.288. Version 17.6.0. 3rd Generation Partnership Project (3GPP), Sept. 2022.

[Air16] Airtel. *Airtel's Open Network*. 2016.

[AK20] A. R. Abdellah and A. Koucheryavy. "VANET traffic prediction using LSTM with deep neural network learning." In: *Internet of Things, Smart Spaces, and Next Generation Networks and Systems: 20th International Conference, NEW2AN 2020*. Springer. 2020, pp. 281–294.

[Ati+23] S. A. Atiiq et al. "Attacks Against Mobility Prediction in 5G Networks." In: *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2023, pp. 1502–1511.

[Big+13] B. Biggio et al. "Evasion attacks against machine learning at test time." In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*. Springer. 2013, pp. 387–402.

[BNL12] B. Biggio, B. Nelson, and P. Laskov. "Poisoning Attacks against Support Vector Machines." In: *Proceedings of the 29th International Coference on International Conference on Machine Learning*. ICML'12. Edinburgh, Scotland: Omnipress, 2012, pp. 1467–1474.

[Cha+08] D. Chakrabarti et al. "Mortal Multi-Armed Bandits." In: *Advances in Neural Information Processing Systems*. Ed. by D. Koller et al. Vol. 21. Curran Associates, Inc., 2008.

[CI22] Capgemini and Intel. *Project Bose - A smart way to enable sustainable 5G networks*. 2022.

[Die+18] T. Diethe et al. "Continual learning in practice." In: *NeurIPS 2018*. 2018.

[DN03] I. Dinur and K. Nissim. "Revealing information while preserving privacy." In: *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2003, pp. 202–210.

[Ekm+08] F. Ekman et al. "Working Day Movement Model." In: *Proceedings of the 1st ACM SIGMOBILE Workshop on Mobility Models*. Hong Kong, Hong Kong, China, 2008, pp. 33–40.

[EMH19] T. Elsken, J. H. Metzen, and F. Hutter. "Neural Architecture Search: A Survey." In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21.

[Eri+20] N. Erickson et al. "AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data." In: *arXiv preprint arXiv:2003.06505* (2020).

[Eri20] Ericsson GAIA. *A framework for simulating mobility patterns in LTE*. Internal Link. 2020.

[Eur23] European Commission. *Principles of the GDPR*. 2023.

[Feu+18] M. Feurer et al. "Practical automated machine learning for the automl challenge 2018." In: *International Workshop on Automatic Machine Learning at ICML*. 2018, pp. 1189–1232.

[Feu+22] M. Feurer et al. “Auto-sklearn 2.0: Hands-free automl via meta-learning.” In: *The Journal of Machine Learning Research* 23.1 (2022), pp. 11936–11996.

[Gho+14] L. Ghouti et al. “Mobility Prediction Using Fully-Complex Extreme Learning Machines.” In: *ESANN*. 2014.

[Gij+22] P. Gijsbers et al. *AMLB: an AutoML Benchmark*. 2022.

[GSS14] I. J. Goodfellow, J. Shlens, and C. Szegedy. “Explaining and harnessing adversarial examples.” In: *arXiv preprint arXiv:1412.6572* (2014).

[Gu+19] T. Gu et al. “BadNets: Evaluating Backdooring Attacks on Deep Neural Networks.” In: *IEEE Access* 7 (2019), pp. 47230–47244.

[HZC21] X. He, K. Zhao, and X. Chu. “AutoML: A survey of the state-of-the-art.” In: *Knowledge-Based Systems* 212 (2021), p. 106622.

[HZS06] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. “Extreme learning machine: Theory and applications.” In: *Neurocomputing* 70.1 (2006). Neural Networks, pp. 489–501.

[Imt+18] S. Imtiaz et al. “Random forests resource allocation for 5G systems: Performance and robustness study.” In: *Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE. 2018.

[Kar+21] S. K. Karmaker (“Santu”) et al. “AutoML to Date and Beyond: Challenges and Opportunities.” In: *ACM Comput. Surv.* 54.8 (Oct. 2021).

[KK10] H. Kaaniche and F. Kamoun. “Mobility prediction in wireless ad hoc networks using neural networks.” In: *arXiv preprint arXiv:1004.4610* (2010).

[Liu+15a] J. Liu et al. “Cloning 3G/4G sim cards with a pc and an oscilloscope: lessons learned in physical security.” In: *BlackHat USA* (2015).

[Liu+15b] J. Liu et al. “Small Tweaks Do Not Help: Differential Power Analysis of MILENAGE Implementations in 3G/4G USIM Cards.” In: *Computer Security – ESORICS 2015*. Cham: Springer International Publishing, 2015, pp. 468–480.

[Liu+18a] Y. Liu et al. “Trojaning attack on neural networks.” In: *25th Annual Network And Distributed System Security Symposium (NDSS 2018)*. Internet Soc. 2018.

[LL24] K. Li and F. Li. “Multi-Fidelity Methods for Optimization: A Survey.” In: *arXiv preprint arXiv:2402.09638* (2024).

[Man+23] J. Manocha et al. *Accelerating the adoption of AI in programmable 5G networks*. 2023.

[MM19] S. Mirjalili and S. Mirjalili. "Genetic algorithm." In: *Evolutionary Algorithms and Neural Networks: Theory and Applications* (2019), pp. 43–55.

[Ora22] Oracle. *Oracle Communications Network Data Analytics Function (OC-NWDAF)*. 2022.

[OV23] A. Oprea and A. Vassilev. *Adversarial machine learning: A taxonomy and terminology of attacks and mitigations (draft)*. Tech. rep. National Institute of Standards and Technology, 2023.

[Ozt+19] M. Ozturk et al. "A novel deep learning driven, low-cost mobility prediction approach for 5G cellular networks: The case of the Control/Data Separation Architecture (CDSA)." In: *Neurocomputing* 358 (2019), pp. 479–489.

[Pav18] B. Pavlyshenko. "Using Stacking Approaches for Machine Learning Models." In: *2018 IEEE Second International Conference on Data Stream Mining & Processing (DSMP)*. 2018, pp. 255–258.

[Ruk16] Rukmini. *India walks to work: Census*. 2016.

[Sha+15a] B. Shahriari et al. "Taking the human out of the loop: A review of Bayesian optimization." In: *Proceedings of the IEEE* 104.1 (2015), pp. 148–175.

[Sho+17] R. Shokri et al. "Membership inference attacks against machine learning models." In: *2017 IEEE symposium on security and privacy (SP)*. IEEE. 2017, pp. 3–18.

[SJ22] V. K. Singh and K. Joshi. "Automated Machine Learning (AutoML): an overview of opportunities for application and research." In: *Journal of Information Technology Case and Application Research* 24.2 (2022), pp. 75–85. eprint: https://doi.org/10.1080/15228053.2022.2074585.

[ST19b] G. Solmaz and D. Turgut. "A Survey of Human Mobility Models." In: *IEEE Access* 7 (2019), pp. 125711–125731.

[Sze+13] C. Szegedy et al. "Intriguing properties of neural networks." In: *arXiv preprint arXiv:1312.6199* (2013).

[Tim21] F. Times. *Closing the AI skills gap*. 2021.

[Wan+21b] C. Wang et al. "FLAML: A Fast and Lightweight AutoML Library." In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 434–447.

[WWH21] Q. Wu, C. Wang, and S. Huang. “Frugal optimization for cost-related hyperparameters.” In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 12. 2021, pp. 10347–10354.

Catching Common Vulnerabilities with Code Language Models

Abstract

Code Language Model (code-LM)-based vulnerability detection for C/C++ faces a substantial challenge. Previous research has shown that even though code-LM-based is better than any prior machine learning approach, they still struggle to generalize well, as shown by the low F1 score. Prior works treated the problem as a binary classification: either vulnerable or non-vulnerable. Looking deeper at the various vulnerability types, we see that this oversimplifies the problem, as different vulnerabilities have different characteristics. This paper investigates the same problem but with a different question, i.e., how would the model perform if the task is to classify whether the code is vulnerable to a specific type? We use the recently released PrimeVul dataset as a basis to investigate the ability to correctly classify different types of vulnerabilities. We conduct two experiments by fine-tuning code LMs on (1) datasets specific to each of the most common types and (2) cumulative datasets incorporating an increasing number of the most common types. We show that it is challenging to correctly identify a specific class of vulnerability in a dataset containing all types of vulnerabilities. However, if the task is modified to correctly identify the most common vulnerabilities, the cumulative model outperforms the previous results using binary classification on the dataset. This result shows a promising path to make code-LM practical in

Syafiq Al Atiiq, Christian Gehrmann, Karim Khalil, Kevin Dahlén. "Catching Common Vulnerabilities with Code Language Models". Submitted to 20^{th} *International Conference on Availability, Reliability and Security, ARES 2025, Ghent, Belgium*.

assisting developers with vulnerability detection tasks in C/C++ code.

1 Introduction

Software vulnerability detection is a resource-intensive task due to the complexity and diversity of software systems [Lin+20]. Crowdstrike 2024 State of Application Security Report[1] mentioned that *"traditional security reviews are time-consuming and expensive"*. Vulnerabilities can emerge from different sources, i.e., design flaws, implementation errors, or misconfigurations. Traditional techniques, such as static analysis [CM04] and dynamic testing [Bal99c], often struggle to keep pace [Zho+24] with the evolving landscape and the increasing software complexity.

Recent advancements in transformers [Vas+17], particularly in code language models (code LMs), have demonstrated impressive abilities in comprehending both natural languages [Bro+20] and programming languages [Wei+24; Guo+24a; Zhe+24]. This has led to the consideration of using code LMs for vulnerability detection. By training code LMs on vast amounts of code (vulnerable and non-vulnerable), code LMs can potentially learn patterns and common vulnerabilities. Unfortunately, the outcome has not been as favorable as one would have hoped.

A recent work, Diversevul [Che+23b], has shown a promising result by fine-tuning smaller code LMs on a curated dataset of ∼330k non-vulnerable and ∼19k vulnerable functions, resulting in improvements over the traditional Graph Neural Network (GNN) [Sca+09] approach. However, even if code LMs are better than GNNs, they still struggle to generalize well. Its best-finetuned model (CodeBERT-based [Fen+20b]) achieved only an 11.94% F1 score, with an unclear cause.

A follow-up work, PrimeVul [Din+25], is introduced as a new benchmark dataset designed to show the limitations of the existing works. It features high-quality data and rigorous de-duplication. It reveals an even deeper problem: state-of-the-art (SOTA) LMs with billions of parameters struggle to generalize, with performance significantly lower than reported. For instance, a StarCoder3 7B model [Li+23] achieved an impressive F1 score of 68.26% on the BigVul [Fan+20] dataset but plummeted to a mere 3.09% when tested on PrimeVul. The result also shows no correlation between parameter size and performance.

The *de-facto* approach used by PrimeVul and DiverseVul (and numerous earlier works on ML-based vulnerability detection [Tha+22; Zho+19b; Zho+24; Fan+20; Cha+22a]) is that they treat the problem with a binary classification approach where multiple vulnerability types are classified into a single label. We hypothesize that this is overly simplistic, mainly for two reasons: i) Each type has its characteristics, and ii) different types have different numbers of representations in real-world data (some vulnerabilities are more common than others). The question from previous works is always asked, i.e., is the code vulnerable? In this paper,

[1] https://www.crowdstrike.com/2024-state-of-application-security-report/

we investigate a different question, i.e., is it vulnerable to a specific type of vulnerability? If it is indeed oversimplistic, then one can achieve a practical performance if one narrows down the detection scope. Another implication is that if one incorporates more vulnerability types into the dataset, the detection performance will decrease as the types increase. We perform extensive vulnerability detection experiments to evaluate our hypothesis addressing the following two research questions:

- **RQ1**: How do different code-LMs perform for each of the most common vulnerability types?
- **RQ2**: How does code-LM performance change when incrementally including more of the most common vulnerability types in the dataset?

To keep the investigation size feasible, we pick the ten most common vulnerability types based on the Common Weakness Enumeration (CWE).

Contributions. We investigate code vulnerability detection from a new angle. Instead of the *de-facto* binary classification approach, we focus on whether a given code snippet is vulnerable to a specific type of vulnerability. This more granular approach reveals new insights: (RQ1) Narrowing down to a single or a small subset of vulnerability types improves per-type detection quality, with some vulnerabilities (i.e., CWE-119, CWE-20) showing patterns that generalize better across other vulnerability types, whereas others (i.e., CWE-190, CWE-703) remain isolated and harder to detect beyond their own category. However, we also demonstrate that it is particularly challenging to correctly identify a specific class of vulnerability in a dataset containing all types of vulnerabilities. (RQ2) Gradually increasing the number of targeted vulnerability types expands the model's coverage but reduces per-type detection quality, ultimately showing a trade-off between specialization and generalization. Our analysis also demonstrates that a model trained on the top 10 most common CWEs, chosen based on their prevalence, outperforms the SOTA results reported by PrimeVul (in terms of F1-score) when evaluated in a similar fashion, highlighting that careful selection and training on a more focused set of vulnerability types can surpass baselines that combine all types into a single label.

Open Science. To support open science and encourage reproducibility, we have made all materials from our study publicly available on GitHub. This includes the dataset, scripts for data processing, model training, evaluation, and the complete experimental results. These resources can be accessed at the following link: `https://github.com/syafiq/commonvul`.

Paper Organization. The rest of the paper is organized as follows. We present the related work in section 2. Then, we discuss the data processing, used code LMs, and the fine-tuning setup in section 3, followed by discussing RQ1 and RQ2 in section 4. We discuss the threats to the validity of our study in section 5. And finally, we draw our conclusion and anticipate future works in section 6.

2 Related Work

2.1 Machine Learning Based Detection

Static Application Security Testing (SAST) has long been a method for identifying vulnerabilities and ensuring code quality. Traditional SAST techniques rely on rule-based systems and heuristics. However, with the birth of machine learning, researchers have explored the application of these techniques to enhance the effectiveness and efficiency of SAST. While SAST tools are widely used in industry, recent studies have shown that they may be less effective [Zho+24].

On the machine learning side, early work [Neu+07] proposed using support vector machine (SVM) [CV95] to predict vulnerable software components by extracting features from code based on historical data. VulDeePecker [Li+18b] advanced this approach by applying deep learning techniques, specifically long short-term memory (LSTM) [HS97] networks, to learn the patterns of insecure code. They treated code as a sequence of tokens and trained the LSTM model to predict the presence of vulnerabilities.

Building upon these, more recent work has focused on leveraging Graph Neural Networks (GNNs) [Sca+09] to incorporate code structure information. Devign [Zho+19b] and ReVeal [Cha+22a] utilized GNNs with code property graphs to identify vulnerabilities based on the structural and semantic properties of code. Similarly, VulChecker [Mir+23] proposed using an enriched program dependence graph for vulnerability detection. Another approach by [Yam+14] modeled code as a graph using GNNs to capture relationships between code elements for identifying vulnerabilities. Before the widespread use of pre-trained language models, early deep learning approaches for vulnerability detection primarily utilized Bidirectional Long Short-Term Memory (BiLSTM) and Bidirectional Gated Recurrent Unit (BiGRU) [Cho+14] networks. These variants of recurrent neural networks (RNN) can process sequential data, such as code tokens, in both forward and backward directions, enabling them to learn the patterns. Notable examples include VulDeePecker [Li+18b] and SySeVR [Li+22].

2.2 Language Model Based Detection

Recent advancements in natural language processing (NLP) have led to the adoption of pre-trained LMs, such as Bidirectional Encoder Representations from Transformers (BERT) [Dev+19b], in the field of code analysis. CodeBERT [Fen+20b] is a pre-trained code LM that adapts BERT to capture the semantic and syntactic properties of code. While CodeBERT is not specifically designed for vulnerability detection, its ability to understand code semantics suggests its potential application in this domain. Diversevul [Che+23b] conducted a comprehensive study on 11 model architectures from 4 families: GNNs [Sca+09], RoBERTa [Liu+20], GPT-2 [Rad+19], and T5 [Raf+20]. Their results demonstrated that code LMs like CodeT5 [Wan+21c; Le+22] and NatGen [Cha+22c],

Dataset	Base	Acc ↑	F1 ↑	Prec ↑	Rec ↑	FPR ↓
DiverseVul [Che+23b]	CT5	94.91	9.39	13.35	7.24	1.78
	CB	94.19	11.94	13.34	10.80	2.65
PrimeVul [Din+25]	CT5	96.67	19.7	-	-	-
	CB	96.87	20.86	-	-	-
	UC	96.86	21.43	-	-	-
	SC2	97.02	18.05	-	-	-
	CG2.5	96.65	19.61	-	-	-

Table 1: Performance of SOTA Vulnerability Detector. CT5: CodeT5, CB: CodeBert, UC: UnixCoder, SC2: StarCoder2, CG2.5: CodeGen2.5. Numbers are taken from the original papers; (-) means no available data.

outperformed the SOTA GNN model as the training data size increased. However, even with better performance than GNNs when trained on larger datasets, DiverseVul confirmed that code LMs still struggle to achieve high performance on the vulnerability detection task. When the models were tested on unseen projects that were not represented in the training data, performance dropped significantly. For example, CodeT5's F1 score fell to only 9.39% on unseen projects. This suggests the models may be overfitting to project-specific patterns and have difficulty generalizing to new codebases. The root causes of these performance limitations remain unclear based on the experiments in DiverseVul. The authors highlight investigating and addressing this generalization challenge to unseen projects as an important direction for future work.

As a follow-up, PrimeVul [Din+25] is introduced. It has rigorous data deduplication and chronological data splitting strategies to mitigate leakage. It presents a new evaluation metric, the Vulnerability Detection Score (VD-S), and a pair-wise evaluation method to assess the model's ability to distinguish between vulnerable code and its fixed counterpart. The authors show that current benchmarks overstate the performance of these models in practical situations. It concludes that even with efforts to improve performance through advanced training techniques and larger parameters, these models still significantly lack the capability for reliable vulnerability detection in real-world scenarios.

2.3 De-facto Approach for Existing Detection

The *de-facto* approach used in the studies from sections 2.1 and 2.2 treats all vulnerability types under the same label. Table 1 compares the performance of several SOTA code LMs on the DiverseVul and PrimeVul datasets. The models evaluated include: CodeT5 (CT5) [Wan+21c] 60M, CodeBERT (CB) [Fen+20b] 125M, UniXcoder (UC) [Guo+22] 125M, StarCoder2 (SC2) [Li+23] 7B, CodeGen2.5 (CG2.5) [Nij+23] 7B. As shown in Table 1, even these advanced models with billions of parameters struggle to achieve high F1 scores, with UnixCoder reaching

Dataset	$\|V\|$	$\|NV\|$	*Source*
DiverseVul	18945	330492	Crawling
PrimeVul-v0.1	6004	218529	BigVul [Fan+20], CrossVul [Nik+21a], CVEFixes [BNM21b], DiverseVul [Che+23b]

Table 2: Previous Datasets.

the highest score of only 21.43% on PrimeVul.

3 Data, Model, and Fine-tuning Setup

Before we explore the two research questions, we explain how we process the data, the base code LMs we use, and the fine-tuning setup for our investigation.

3.1 Data

Previous data

As shown in Table 2, the dataset contains the following details: DiverseVul is quite large, containing $>$ 300k functions. However, PrimeVul identified some key limitations with DiverseVul. First, the labels are noisy, with only 60% accuracy for vulnerable functions. Second, even after applying hash-based deduplication, DiverseVul still has 3.3% duplicate functions.

While DiverseVul has a larger raw number of samples, PrimeVul is better suited for our purpose. The label accuracy in PrimeVul is much higher at 92%. This is important as noisy labels can harm model training. Additionally, PrimeVul takes care to de-duplicate the data, avoiding potential data leakage between train and test sets that could inflate performance. So even though PrimeVul has fewer total functions ($\sim$200k), the higher quality of the data is more valuable than simply having a larger quantity of noisy samples.

This paper uses PrimeVul-v0.1[2]. This updated version of PrimeVul was created and released by the original authors of the PrimeVul dataset. Compared to the version described in the original PrimeVul paper [Din+25], PrimeVul-v0.1 includes additional metadata, specifically comprehensive CWE annotations that were not present in the initial release. Apart from using this updated version, we have made no modifications to the dataset. Thus, PrimeVul-v0.1 retains all the key benefits of the original data, such as high-quality labels and thorough de-duplication, while providing the CWE information essential for our investigation. For brevity, when we mention PrimeVul, we refer to PrimeVul-v0.1.

[2]https://github.com/DLVulDet/PrimeVul

Data preprocessing

We take the vulnerable samples and show them in the v column in Table 3. As shown, the data is significantly skewed, with some CWEs being more prevalent in the dataset while others have representations as low as 1-2 samples per CWE. We initiated the splitting process by defining a list of the top 10 most prevalent CWEs: CWE_{119}, CWE_{125}, CWE_{787}, CWE_{20}, CWE_{476}, CWE_{200}, CWE_{416}, CWE_{703}, CWE_{190}, CWE_{399}. We also calculated the overall ratio of vulnerable (v) to non-vulnerable (nv) entries, represented as r. In PrimeVul, r is equal to 36. For each vulnerable entry in the list of CWEs, we randomly sampled a number of non-vulnerable counterparts equal to the size of CWE_i multiplied by the ratio r. This ensures controlled experiments for consistent comparison across CWEs and maintains class distribution during the investigation of CWE_i to preserve interpretability. Finally, we split the dataset into training and testing sets using a 90:10 ratio.

Dataset formation

The released PrimeVul contains already split data: train, validation, and test. Let the merged PrimeVul be denoted as D, obtained by combining the three subsets: D_{train}, $D_{validation}$, and D_{test}.

$$D = D_{train} \cup D_{validation} \cup D_{test} \tag{1}$$

For each CWE i, we create a CWE-specific dataset D_i by identifying the vulnerable entries V_i and randomly sampling non-vulnerable counterparts NV_i. The number of non-vulnerable samples is determined by multiplying the number of vulnerable functions for each CWE by a factor r.

$$|NV_i| = r \times |V_i| \tag{2}$$

While this approach standardizes the interpretability of our results, it also ensures that no non-vulnerable sample is reused across different CWE subsets. We maintain a hash table to track selected entries, ensuring that once a sample is chosen for a particular CWE dataset, it cannot be selected again for another. Consequently, this procedure preserves the diversity of non-vulnerable samples and prevents overlap between different experiments. In the context of our study, the focus remains on understanding CWE-specific vulnerabilities rather than replicating real-world class distributions, making this controlled sampling strategy suitable for our purposes.

The CWE-specific dataset D_i is then formed by combining the vulnerable and non-vulnerable samples: $D_i = V_i \cup NV_i$. For CWEs with more than two samples, we split each CWE-specific dataset D_i into train and test sets, $D_{i,train}$ and $D_{i,test}$, using a 90:10 ratio with stratified sampling to preserve the vulnerable

CWE_i	$\|V\|$	$\|NV\|$	$\|D_{i,train}\|$	$\|D_{i,test}\|$	Top n	$\|D^{cum}_{train,n}\|$	$\|D^{cum}_{test,n}\|$
119	719	25884	23942	2661	-	-	-
125	677	24372	22543	2506	2	46485	5167
787	504	18144	16782	1866	3	63267	7033
20	471	16956	15683	1744	4	78950	8777
476	373	13428	12420	1381	5	91370	10158
200	321	11556	10688	1189	6	102058	11347
416	295	10620	9823	1092	7	111881	12439
703	257	9252	8557	952	8	120438	13391
190	228	8208	7592	844	9	128030	14235
399	196	7056	6526	726	10	134556	14961
1–2 samples/CWE	55	1980	-	2035	-	-	2035
The rest	1908	67073	60365	7118	-	-	7118
$\sum$	6004	218529	-	$\|D^{comp}_{test,i}\|$ = 24114	-	-	$\|D^{cum}_{test,all,n}\|$ = 24114

Table 3: Breakdown of the produced subset within PrimeVul.

to non-vulnerable ratio where $V_{i,train}$, $NV_{i,train}$, $V_{i,test}$, and $NV_{i,test}$ represent the vulnerable and non-vulnerable samples in the train and test sets, respectively:

$$D_i = D_{i,train} \cup D_{i,test}, ; |D_{i,train}| = 0.9 \times |D_i|, ; |D_{i,test}| = 0.1 \times |D_i| \quad (3)$$

$$\frac{|V_{i,train}|}{|NV_{i,train}|} = \frac{|V_{i,test}|}{|NV_{i,test}|} = \frac{|V_i|}{|NV_i|} \quad (4)$$

For CWEs with two or fewer samples, we create a test dataset that includes all available positive samples and a randomly sampled set of negative samples (with the same ratio r to the positive samples). This approach is necessary since stratified splitting becomes impractical with such few samples. As a consequence, these CWEs (55 entries in total) are only present in the test dataset and not in the training dataset.

To evaluate the generalization capabilities of our models, we create a comprehensive test dataset $D_{test,all}$ by merging the test sets of all CWEs, including the top 10 CWEs (CWE_{top10}) and the remaining CWEs (CWE_{rest}).

$$D_{test,all} = \bigcup_{i \in CWE_{top10} \cup CWE_{rest}} D_{i,test} \quad (5)$$

Additionally, we create CWE-specific comprehensive test datasets $D^{comp}_{test,i}$ by relabeling the samples in the comprehensive test dataset based on whether they belong to the specific CWE i or not.

$$D^{comp}_{test,i} = (x, y_i) | x \in D_{test,all}, y_i = \mathbb{1}(x \in D_{i,test}) \quad (6)$$

where $\mathbb{1}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise. Note that $D^{comp}_{test,i}$ differs from $D_{i,test}$ in that it includes samples from all CWEs, with labels indicating whether they belong to CWE i or not, while $D_{i,test}$ only contains samples from CWE i.

Furthermore, we create cumulative datasets to assess the performance of models trained on an increasing number of CWEs. Let $CWE_{top10} = cwe_1, cwe_2, ..., cwe_{10}$ be the set of the top 10 most frequent CWEs. For each $n \in 1, 2, ..., 10$, we create a cumulative training dataset $D^{cum}_{train,n}$ and a cumulative testing dataset $D^{cum}_{test,n}$ by merging the train and test sets of the top n CWEs, respectively.

$$D^{cum}_{train,n} = \bigcup_{i=1}^{n} D_{cwe_i,train} \quad (7)$$

Model Name	Abbr.	Params	Arch.	Approach
CodeT5 [Wan+21c]	CT5	60M	Enc-Dec	Fine-tuning
CodeBERT [Fen+20b]	CB	125M	Encoder	Fine-tuning
UnixCoder [Guo+22]	UC	125M	Encoder	Fine-tuning

Table 4: The three pre-trained code language models compared in this research.

$$D^{cum}_{test,n} = \bigcup_{i=1}^{n} D_{cwe_i,test} \tag{8}$$

Note that $D^{cum}_{train,n}$ and $D^{cum}_{test,n}$ differ from $D_{i,train}$ and $D_{i,test}$ in that they represent the cumulative datasets for the top n CWEs, while $D_{i,train}$ and $D_{i,test}$ represent the train and test sets for an individual CWE i.

Lastly, we create comprehensive test datasets $D^{cum}_{test,all,n}$ for each cumulative set of top n CWEs by relabeling the samples in the comprehensive test dataset $D_{test,all}$ based on whether they belong to any of the top n CWEs or not.

$$D^{cum}_{test,all,n} = (x, y_n) | x \in D_{test,all}, y_n = \mathbb{1}(x \in \bigcup_{i=1}^{n} D_{cwe_i,test}) \tag{9}$$

This approach allows us to investigate the trade-off between specialization and generalization in vulnerability detection as the number of CWEs considered grows. The final data for the top 10 most prevalent CWEs, the cumulative datasets, and the rest of the CWEs can be seen in Table 3.

3.2 Code Language Models Studied

This study compares the effectiveness of using CWE-specific and cumulative subsets of training data on three pre-trained code LMs. The models, summarized in Table 4, include CodeT5 (CT5), a 60 million parameter encoder-decoder model, and CodeBERT (CB) and UnixCoder (UC), both 125 million parameter encoder-only models. We focus our analysis on these three smaller models used in the PrimeVul [Din+25] and do not include experiments with larger models like StarCoder2 [Li+23] and CodeGen2.5 [Nij+23]. The main reasons for this are threefold.

First, these larger models still struggle to perform well on the realistic vulnerability detection task despite their substantial size. Second, the model with the highest performance in the PrimeVul study is not the model with the most parameters. Third, our evaluation will be extensive, covering multiple CWEs and sub-datasets per model. Resource-wise, including the larger models, would consume a significant amount of computational resources. Therefore, we focus on the three models used in PrimeVul to balance comprehensive evaluation and feasibility given resource constraints. All three models underwent fine-tuning on all the

subsets. Their performance was then evaluated separately to assess the impact of the specialized training data.

3.3 Fine-tuning Setup

Our experimental setup for fine-tuning the language models draws from the methodologies established in prior benchmark studies [Che+23b; Din+25]. We employ the AdamW optimizer with a learning rate of 2×10^{-5} and train for ten epochs. To fill the available memory, we use a batch size of 32 per GPU and accumulate gradients over eight steps, simulating a larger batch size. The learning rate schedule incorporates a linear warmup phase of 1000 steps, followed by a linear decay. Our experiments are conducted on a high-performance computing (HPC) environment managed by the Slurm[3] workload manager [JW23]. This cluster provides multiple NVIDIA A100 GPUs, enabling parallelism and efficient resource allocation during fine-tuning.

4 Experiment and Discussion

4.1 Parameters for Discussion

Metric	Formula	Description
Precision	$\frac{TP}{TP+FP}$	Proportion of correctly predicted vulnerabilities among all predictions; measures false alarm avoidance.
Recall	$\frac{TP}{TP+FN}$	Proportion of actual vulnerabilities successfully detected; critical for security coverage.
F1-Score	$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$	Harmonic mean of precision and recall; balances detection coverage and accuracy.
FPR	$\frac{FP}{FP+TN}$	False Positive Rate; Rate of falsely flagging non-vulnerable code.
TNR	$\frac{TN}{TN+FP}$	True Negative Rate; Rate of correctly identifying non-vulnerable code.

Table 5: Evaluation metrics used in our study.

Table 5 shows the metrics we use to analyze the results. Here, TP, FP, FN, and TN represent the number of true positives, false positives, false negatives, and true negatives, respectively. These metrics are standard in binary classification tasks, and we include them for completeness and ease of interpretation.

[3] https://slurm.schedmd.com/overview.html

4.2 RQ1-Performance on each of the top 10 CWE

Let M_i represent the model trained to classify CWE i where $i \in 119, 125, 787, 20, 476, 200, 416, 703, 190, 399$ corresponding to the top 10 CWEs. In Table 6, M_i's performance metrics are evaluated on $D_{i,test}$. For Table 7, the performance metrics are evaluated on $D^{comp}_{test,i}$. Additionally, in Table 7, we introduce two new metrics: TP' and $F1'$ (described later in equation 10). TP' measures the number of true positives the model identifies when evaluated on the full test set $D_{test,all}$, which includes vulnerabilities from all CWEs, not just the specific CWE i the model was trained on. This captures the model's ability to detect vulnerabilities beyond its primary focus. $F1'$ is the $F1$ score calculated using TP' instead of TP. It provides a more comprehensive assessment of the model's performance, taking into account its capacity to generalize and identify vulnerabilities across different CWEs.

Observation 1: Models trained on CWEs with more training data show higher performance metrics.

Models trained on CWEs with larger training sets ($|D_{i,train}|$) generally achieve higher performance metrics (i.e., 119, 125). The abundance of data for these CWEs provides richer patterns for the model to learn, resulting in improved precision, recall, and overall F1 scores.

Observation 2: Precision surpasses recall on single-CWE tests.

When evaluated on a CWE-specific test set $D_{i,test}$, the model exhibits

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i} > \frac{TP_i}{TP_i + FN_i} = \text{Recall}_i.$$

This indicates that, within its trained CWE domain, the model identifies vulnerable samples more conservatively, yielding fewer false alarms (low FP_i) at the expense of missing some vulnerabilities (non-zero FN_i).

Observation 3: False positive rate rises on comprehensive tests.

On the combined test set $D^{comp}_{test,i}$, which includes multiple CWEs, the model's false positive rate

$$\text{FPR}_i = \frac{FP_i}{FP_i + TN_i}$$

increases significantly compared to $D_{i,test}$. Although TP_i remains unchanged, the presence of vulnerabilities from other CWEs leads to more FP_i, thereby reducing precision:

$$\text{Precision}_i(D^{comp}_{test,i}) = \frac{TP_i}{TP_i + FP_i} \downarrow$$

CWE	Model	Acc ↑	F1 ↑	Prec ↑	Rec ↑	FPR ↓	TNR ↑	TP ↑	TN ↑	FP ↓	FN ↓
119	CT5	97.93	57.36	64.91	51.39	0.77	99.23	37	2569	20	35
	CB	97.56	54.55	54.93	54.17	1.24	98.76	39	2557	32	33
	UC	97.75	58.90	58.11	59.72	1.20	98.80	43	2558	31	29
125	CT5	97.29	45.16	50.00	41.18	1.15	98.85	28	2410	28	40
	CB	97.65	53.54	57.63	50.00	1.03	98.97	34	2413	25	34
	UC	97.73	55.12	59.32	51.47	0.98	99.02	35	2414	24	33
787	CT5	97.37	19.67	60.00	11.76	0.22	99.78	6	1811	4	45
	CB	97.59	38.36	63.64	27.45	0.44	99.56	14	1807	8	37
	UC	97.48	37.33	58.33	27.45	0.55	99.45	14	1805	10	37
20	CT5	97.25	36.84	50.00	29.17	0.83	99.17	14	1682	14	34
	CB	97.94	55.00	68.75	45.83	0.59	99.41	22	1686	10	26
	UC	97.99	59.77	66.67	54.17	0.77	99.23	26	1683	13	22
476	CT5	97.32	27.45	53.85	18.42	0.45	99.55	7	1337	6	31
	CB	97.76	41.51	73.33	28.95	0.30	99.70	11	1339	4	27
	UC	98.19	57.63	80.95	44.74	0.30	99.70	17	1339	4	21
200	CT5	97.73	50.91	63.64	42.42	0.69	99.31	14	1148	8	19
	CB	97.81	50.00	68.42	39.39	0.52	99.48	13	1150	6	20
	UC	98.23	64.41	73.08	57.58	0.61	99.39	19	1149	7	14
416	CT5	97.62	27.78	83.33	16.67	0.09	99.91	5	1061	1	25
	CB	97.44	30.00	60.00	20.00	0.38	99.62	6	1058	4	24
	UC	97.99	45.00	90.00	30.00	0.09	99.91	9	1061	1	21
703	CT5	97.69	31.25	83.33	19.23	0.11	99.89	5	925	1	21
	CB	97.90	54.55	66.67	46.15	0.65	99.35	12	920	6	14
	UC	97.69	47.62	62.50	38.46	0.65	99.35	10	920	6	16
190	CT5	96.80	22.86	33.33	17.39	0.97	99.03	4	813	8	19
	CB	97.51	43.24	57.14	34.78	0.73	99.27	8	815	6	15
	UC	97.27	34.29	50.00	26.09	0.73	99.27	6	815	6	17
399	CT5	97.80	38.46	83.33	25.00	0.14	99.86	5	705	1	15
	CB	97.52	43.75	58.33	35.00	0.71	99.29	7	701	5	13
	UC	97.66	45.16	63.64	35.00	0.57	99.43	7	702	4	13

Table 6: Different code LMs performance for each of the most common vulnerability types. Trained on $D_{i,train}$ and tested on $D_{i,test}$.

CWE	Model	Acc ↑	F1 ↑	Prec ↑	Rec ↑	FPR ↓	TNR ↑	TP ↑	TN ↑	FP ↓	FN ↓	TP' ↑	F1' ↑
119	CT5	98.37	15.85	9.37	51.39	1.49	98.51	37	23684	358	35	161	29.81
	CB	97.95	13.61	7.78	54.17	1.92	98.08	39	23580	462	33	195	32.83
	UC	97.91	14.60	8.32	59.72	1.97	98.03	43	23568	474	29	230	38.21
125	CT5	97.90	9.96	5.67	41.18	1.94	98.06	28	23580	466	40	118	19.98
	CB	98.60	16.79	10.09	50.00	1.26	98.74	34	23743	303	34	79	15.43
	UC	98.34	14.89	8.71	51.47	1.53	98.47	35	23679	367	33	103	18.92
787	CT5	99.48	8.76	6.98	11.76	0.33	99.67	6	23983	80	45	26	6.74
	CB	99.23	13.08	8.59	27.45	0.62	99.38	14	23914	149	37	36	8.47
	UC	99.00	10.41	6.42	27.45	0.85	99.15	14	23859	204	37	51	11.27
20	CT5	98.74	8.43	4.93	29.17	1.12	98.88	14	23796	270	34	118	24.41
	CB	98.97	15.07	9.02	45.83	0.92	99.08	22	23844	222	26	96	20.62
	UC	98.41	11.93	6.70	54.17	1.50	98.50	26	23704	362	22	172	32.00
476	CT5	99.09	6.01	3.59	18.42	0.78	99.22	7	23888	188	31	77	17.48
	CB	99.43	13.84	9.09	28.95	0.46	99.54	11	23966	110	27	30	7.43
	UC	99.38	18.48	11.64	44.74	0.54	99.46	17	23947	129	21	44	10.56
200	CT5	99.02	10.61	6.06	42.42	0.90	99.10	14	23864	217	19	110	23.91
	CB	99.37	14.61	8.97	39.39	0.55	99.45	13	23949	132	20	57	13.70
	UC	98.83	11.84	6.60	57.58	1.12	98.88	19	23812	269	14	110	22.56
416	CT5	99.39	6.41	3.97	16.67	0.50	99.50	5	23963	121	25	51	12.52
	CB	99.48	8.70	5.56	20.00	0.42	99.58	6	23982	102	24	38	9.56
	UC	99.34	10.17	6.12	30.00	0.57	99.43	9	23946	138	21	50	11.99
703	CT5	99.68	11.63	8.33	19.23	0.23	99.77	5	24033	55	21	25	6.68
	CB	98.81	7.74	4.23	46.15	1.13	98.87	12	23816	272	14	90	18.54
	UC	99.07	8.16	4.57	38.46	0.87	99.13	10	23879	209	16	43	9.49
190	CT5	98.54	2.22	1.18	17.39	1.39	98.61	4	23757	334	19	77	14.98
	CB	99.07	6.64	3.67	34.78	0.87	99.13	8	23881	210	15	71	15.69
	UC	99.37	7.32	4.26	26.09	0.56	99.44	6	23956	135	17	23	5.56
399	CT5	98.97	3.86	2.09	25.00	0.97	99.03	5	23860	234	15	100	21.60
	CB	98.49	3.70	1.96	35.00	1.46	98.54	7	23743	351	13	106	20.29
	UC	98.91	5.07	2.73	35.00	1.03	98.97	7	23845	249	13	122	25.87

Table 7: Similar to Table 6, trained on $D_{i,train}$, but tested on $D^{comp}_{test,i}$. Additional measurement on F1' that is tested on $D_{test,all}$, i.e., keeping the original label of vulnerable entries, not just to specific CWE in the respective row.

Observation 4: Cross-CWE misclassification drives precision decline.

The drop in precision on $D_{test,i}^{comp}$ (compared to $D_{i,test}$) arises because the model misclassifies vulnerabilities from other CWEs as belonging to CWE i. These non-target CWE vulnerabilities increase FP_i, diluting the precision. While the model still identifies CWE-i vulnerabilities well, it cannot sufficiently discriminate them from other vulnerability types in a larger pool.

Observation 5: Introducing $F1'$ to capture generalization.

We also evaluate how well a model trained solely on CWE i detects *any* vulnerability in $D_{\text{test,all}}$. Let TP_i, FP_i, FN_i be the usual counts for CWE i, but note that FP_i can include code vulnerable to other CWEs (which are *true positives* if we treat "any vulnerability" as a positive). Denote by TP' those "other-CWE" samples within FP_i, and by FN' all other-CWE vulnerabilities the model misses. Our revised "any vulnerability" F1 score is

$$F1' \;=\; 2 \cdot \frac{(TP_i + TP')}{2\,(TP_i + TP') \;+\; (\,FP_i - TP'\,) \;+\; (\,FN_i + FN'\,)} \tag{10}$$

This reclassification corrects for cross-CWE detections (TP') and missed other-CWE samples (FN'). A higher $F1'$ indicates that a model, while trained on one CWE, inadvertently captures patterns shared by multiple CWEs.

Reflection on RQ1.

In the context of code-LM-based vulnerability detection, recall is particularly critical. Missing true vulnerabilities (low recall) can have severe security implications. As shown in Tables 6 and 7, recall often declines for CWEs represented by fewer training samples, highlighting the challenges of imbalanced data. Although the F1 score is commonly used, in these cases, it fluctuates significantly, driven mostly by variations in precision rather than by the model's true coverage of vulnerable instances. Consequently, relying solely on F1 in this setting may be misleading, as it can mask the underlying variability in model performance.

To further understand model generalization, we introduced TP' and $F1'$. These metrics measure how well a model trained on one CWE can inadvertently detect vulnerabilities from other CWEs. The results show that certain CWEs generalize more effectively than others:

- **High TP' and F1':** CWEs such as CWE-119 and CWE-20 achieve notably higher TP' and $F1'$, suggesting these vulnerability types share patterns or characteristics that are more universally detectable.

CWE	TP' Range	Transferability	Features
119	162–230	Strong	Overlapping
190	23–77	Low	More unique

Table 8: Transferability analysis using TP'.

- **Low TP' and F1':** In contrast, CWEs like CWE-190 and CWE-703 display low TP' and $F1'$, implying their unique vulnerability patterns are less readily transferred when the model is trained on other CWEs.

Table 8 summarizes the observed range of TP' values and the corresponding insights. For instance, CWE-119 exhibits strong transferability (i.e., its patterns help in detecting unrelated CWEs), whereas CWE-190 shows low transferability, indicating relatively unique vulnerability features.

The variability in TP' and $F1'$ indicates that groups of CWEs may share structural similarities, enabling improved detection across that cluster. Identifying these clusters and prioritizing CWEs with overlapping traits can enhance generalization and detection performance. However, the fluctuations in $F1'$ indicate the need to carefully balance training data among various CWEs. By selectively focusing on transferable CWEs, future research could develop more efficient training strategies that maximize generalizability and real-world utility.

RQ1 Summary. Models trained on a single, common vulnerability type achieve higher recall, and F1 scores for that specific CWE, yet their performance decreases when tested on more diverse sets. In particular, we show that it is challenging to correctly identify a specific class of vulnerability in a dataset containing all types of vulnerabilities, as evidenced by significant drops in precision when models are evaluated on comprehensive test sets. Some CWEs, like CWE-119 or CWE-20, provide patterns that also help with detecting other vulnerabilities, while others (i.e., CWE-190, CWE-703) remain more isolated and do not easily transfer their learned patterns beyond their own type.

4.3 RQ2-Incrementally include more CWEs

Let M_n represent the model trained to classify the top n CWEs where $n \in 2, 3, ..., 10$. The training data for each model M_n is $D^{cum}_{train,n}$. Each M_n is tested on $D^{cum}_{test,n}$ and $D^{cum}_{test,all,n}$, shown in Tables 9 and 10, respectively. Additional measurement on $D_{test,all}$ is also shown in Table 10 showing the TP' and $F1'$. Note that $F1'$ calculation is similar to the equation 10, only replacing i with n.

Observation 1: Non-monotonic decrease in recall as n grows.

As n increases, the model's recall tends to decrease, indicating that expanding the range of CWEs reduces the model's effectiveness in detecting each individual CWE accurately.

Top n CWE	Model	Acc ↑	F1 ↑	Prec ↑	Rec ↑	FPR ↓	TNR ↑	TP ↑	TN ↑	FP ↓	FN ↓
2	CT5	97.17	47.86	47.86	47.86	1.45	98.55	67	4954	73	73
	CB	97.19	49.12	48.28	50.00	1.49	98.51	70	4952	75	70
	UC	97.35	51.25	51.06	51.43	1.37	98.63	72	4958	69	68
3	CT5	97.51	43.73	56.67	35.60	0.76	99.24	68	6790	52	123
	CB	97.21	39.13	48.09	32.98	0.99	99.01	63	6774	68	128
	UC	97.44	43.75	54.26	36.65	0.86	99.14	70	6783	59	121
4	CT5	96.91	42.95	43.22	42.68	1.57	98.43	102	8404	134	137
	CB	97.07	39.24	45.11	34.73	1.18	98.82	83	8437	101	156
	UC	96.72	42.40	40.61	44.35	1.82	98.18	106	8383	155	133
5	CT5	96.96	39.29	43.10	36.10	1.34	98.66	100	9749	132	177
	CB	96.60	38.06	37.86	38.27	1.76	98.24	106	9707	174	171
	UC	96.80	45.74	42.55	49.46	1.87	98.13	137	9696	185	140
6	CT5	97.07	43.15	45.99	40.65	1.34	98.66	126	10889	148	184
	CB	96.47	37.25	36.17	38.39	1.90	98.10	119	10827	210	191
	UC	97.39	44.36	53.15	38.06	0.94	99.06	118	10933	104	192
7	CT5	96.78	38.02	40.07	36.18	1.52	98.48	123	11915	184	217
	CB	97.24	34.67	49.19	26.76	0.78	99.22	91	12005	94	249
	UC	97.37	39.56	53.23	31.47	0.78	99.22	107	12005	94	233
8	CT5	97.27	40.00	50.00	33.33	0.94	99.06	122	12903	122	244
	CB	96.24	34.20	32.75	35.79	2.07	97.93	131	12756	269	235
	UC	96.59	40.73	38.77	42.90	1.90	98.10	157	12777	248	209
9	CT5	96.71	35.18	38.14	32.65	1.49	98.51	127	13640	206	262
	CB	96.35	33.84	33.50	34.19	1.91	98.09	133	13582	264	256
	UC	97.03	38.07	44.22	33.42	1.18	98.82	130	13682	164	259
10	CT5	97.17	40.28	47.51	34.96	1.09	98.91	143	14394	158	266
	CB	96.28	28.17	29.86	26.65	1.76	98.24	109	14296	256	300
	UC	96.41	34.11	34.24	33.99	1.83	98.17	139	14285	267	270

Table 9: Code LM performance changes when incrementally including more of the most common CWEs. Trained on $D^{cum}_{train,n}$, tested on $D^{cum}_{test,n}$.

Top n CWE	Model	Acc ↑	F1 ↑	Prec ↑	Rec ↑	FPR ↓	TNR ↑	TP ↑	TN ↑	FP ↓	FN ↓	TP' ↑	F1' ↑
2	CT5	97.39	17.56	10.75	47.86	2.32	97.68	67	23418	556	73	226	33.92
	CB	97.74	20.47	12.87	50.00	1.98	98.02	70	23500	474	70	172	38.66
	UC	97.66	20.31	12.65	51.43	2.07	97.93	72	23477	497	68	243	38.16
3	CT5	98.14	23.29	17.30	35.60	1.36	98.64	68	23598	325	123	150	37.04
	CB	98.21	22.62	17.21	32.98	1.27	98.73	63	23620	303	128	106	35.29
	UC	98.31	25.55	19.61	36.65	1.20	98.80	70	23636	287	121	151	38.71
4	CT5	97.36	24.29	16.97	42.68	2.09	97.91	102	23376	499	137	199	42.78
	CB	97.97	25.30	19.90	34.73	1.40	98.60	83	23541	334	156	132	39.89
	UC	97.22	24.01	16.46	44.35	2.25	97.75	106	23337	538	133	214	44.32
5	CT5	97.61	25.74	20.00	36.10	1.68	98.32	100	23437	400	177	167	41.38
	CB	97.25	24.23	17.73	38.27	2.06	97.94	106	23345	492	171	196	39.43
	UC	96.99	27.37	18.92	49.46	2.46	97.54	137	23250	587	140	258	44.87
6	CT5	97.55	29.89	23.64	40.65	1.71	98.29	126	23397	407	184	229	43.89
	CB	**97.08**	**25.29**	**18.86**	**38.39**	**2.15**	**97.85**	**119**	**23292**	**512**	**191**	**186**	**40.76**
	UC	**97.90**	**31.76**	**27.25**	**38.06**	**1.32**	**98.68**	**118**	**23489**	**315**	**192**	**217**	**46.02**
7	CT5	97.27	26.37	21.26	34.71	1.84	98.16	118	23337	437	222	188	39.80
	CB	97.92	26.65	26.53	26.76	1.06	98.94	91	23522	252	249	165	40.56
	UC	98.00	30.75	30.06	31.47	1.05	98.95	107	23525	249	233	184	45.00
8	CT5	97.66	31.39	28.29	35.25	1.38	98.62	129	23421	327	237	204	45.36
	CB	96.75	25.07	19.29	35.79	2.31	97.69	131	23200	548	235	189	38.49
	UC	97.03	30.46	23.61	42.90	2.14	97.86	157	23240	508	209	224	44.23
9	CT5	97.22	27.46	23.69	32.65	1.72	98.28	127	23316	409	262	189	41.73
	CB	96.96	26.60	21.77	34.19	2.01	97.99	133	23247	478	256	186	40.44
	UC	97.57	30.70	28.38	33.42	1.38	98.62	130	23397	328	259	180	44.56
10	**CT5**	**97.57**	**32.84**	**30.95**	**34.96**	**1.35**	**98.65**	**143**	**23386**	**319**	**266**	**216**	**47.14**
	CB	96.98	23.04	20.30	26.65	1.81	98.19	109	23277	428	300	143	37.24
	UC	96.94	27.36	22.90	33.99	1.97	98.03	139	23237	468	270	186	41.97

Table 10: Similar to Table 9, trained on $D^{cum}_{train,n}$, but tested on $D^{cum}_{test,all,n}$. Additional measurement on F1' that is tested on $D_{test,all}$, i.e., keeping the original label of vulnerable entries, not just to specific top n in the respective row.

$$n \uparrow \implies \text{Recall}_n(D^{cum}_{test,n}), \text{Recall}_n(D^{cum}_{test,all,n}) \downarrow .$$

However, the recall does not decrease strictly monotonically. For example, recall may rise when moving from $n = 3$ to $n = 4$ CWEs or from $n = 5$ to $n = 6$. This can occur because adding more CWEs sometimes introduces patterns that help the model recognize previously missed vulnerabilities, offsetting some of the lost specificity. Nonetheless, the overarching trend remains that expanding the model's coverage often dilutes its per-CWE detection power.

Observation 2: Fluctuations in precision and FPR on $D^{cum}_{test,n}$.

When tested on $D^{cum}_{test,n}$, as n increases, both precision and FPR can vary significantly. Changes in the distribution of vulnerability types and overlapping features across CWEs can introduce instability in the model's decision boundaries, causing precision and FPR to rise or fall non-linearly.

Observation 3: Improved precision but still varied FPR.

When tested on $D^{cum}_{test,all,n}$, as n increases, models trained on more CWEs show relatively higher precision (due to higher TP) as n becomes large, even though FPR is still varied. Formally, we observe:

$$n \uparrow \implies \text{Precision}_n(D^{cum}_{test,all,n}) \uparrow \text{ and } \text{TP}_n(D^{cum}_{test,all,n}) \uparrow .$$

This trend occurs because training on more CWEs allows the model to recognize a wider range of vulnerability patterns, leading to more TP and improved precision when tested on a comprehensive dataset. However, accommodating diverse CWEs also introduces complexity in the model's decision boundaries, causing fluctuations in FPR. The improved precision due to improved TP affects the F1 and F1' scores in a similar way, as shown in Figure 1.

Figure 1 illustrates the performance trajectory as n increases. Both F1 and F1' scores generally increase, albeit with non-monotonic fluctuations. This pattern suggests that expanding the range of vulnerability types in training typically enhances detection capabilities, though with variations that reflect the complex interactions between different vulnerability patterns. Note that the gap between F1 and F1' stems from their different evaluation contexts: F1 is calculated on test sets $D^{cum}_{test,all,n}$ while F1' is calculated on $D_{test,all}$. When our model identifies a vulnerability outside its explicit training scope, this counts as a false positive in the F1 calculation but as a true positive in the F1' calculation.

Observation 4: Comparing FPR and TNR across different evaluations.

We note that testing on a broader set $D^{cum}_{test,all,n}$ elevates the FPR and reduces the TNR relative to testing on $D^{cum}_{test,n}$ alone. While in a controlled environ-

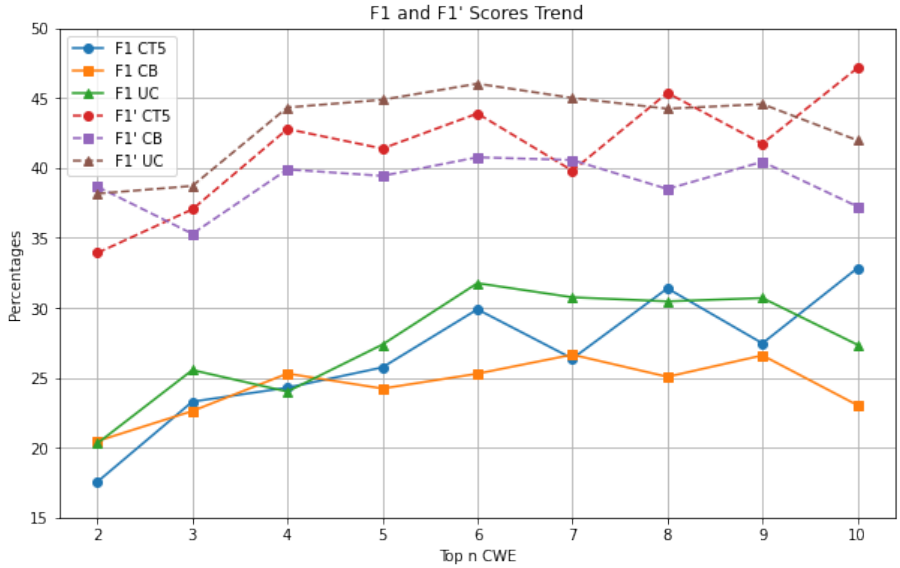

Figure 1: F1 and F1' trend based on Table 10.

ment (fewer CWEs, narrower test set), the model can maintain tight, well-defined decision boundaries, exposure to a wider spectrum of vulnerabilities blurs these boundaries and increases confusion. Consequently, although generalist models handle diverse vulnerabilities more gracefully, they still face increased challenges in maintaining low false positives compared to the narrower, per-CWE tests.

Observation 5: Balancing specialization and generalization.

When evaluated on $D^{cum}_{test,all,n}$, increasing n leads to a trade-off between specialization and generalization:

- Specialized models (small n): Tend to have a higher recall for a limited set of CWEs but may suffer from lower precision when encountering unfamiliar vulnerability patterns.
- Generalist models (large n): Achieve improved precision across a broad range of vulnerabilities, but at the cost of lower recall for any individual CWE.

The appropriate balance depends on the specific priorities of the vulnerability detection setting, whether it is more critical to minimize missed vulnerabilities (favoring recall) or reduce false alarms (favoring precision).

Observation 6: Cumulative models outperform PrimeVul in comparative metrics.

Although the F1 metric has limitations for evaluating vulnerability detection, primarily due to its balance between precision and recall, which can be skewed by how well a model identifies non-vulnerable code, it remains necessary for comparative analysis since PrimeVul does not provide detailed performance metrics for recall, TP, and FN. However, a direct comparison of our standard F1 scores with PrimeVul's reported metrics would be inappropriate since our experiments focus specifically on the top 10 CWEs while PrimeVul contains all vulnerability types. Instead, F1' provides a more suitable comparative benchmark as it evaluates performance across all vulnerabilities in $D_{test,all}$.

As shown in bold from Table 10, our best models achieve substantially higher F1' scores compared to PrimeVul's reported F1 scores in Table 1: CT5 47.14% when $n = 10$ (versus PrimeVul's 19.7%), CB 40.76% when $n = 6$ (versus PrimeVul's 20.86%), UC: 46.02% when $n = 6$ (versus PrimeVul's 21.43%). It is important to note that methodological differences exist (as described in section 3.1), our stratified sampling versus PrimeVul's original set, which may influence these comparisons. Nevertheless, the consistently higher F1' scores indicate that models trained on common CWEs can successfully detect other vulnerability types, suggesting shared patterns across different CWE categories.

Reflection on RQ2.

When we compare the model trained cumulatively on multiple CWEs (M_n) to the aggregate results of separate models individually trained on each CWE ($\sum M_i$), the cumulatively trained model generally identifies more true positives (or fewer false negatives as the implication). For instance, the CT5 model trained on the top four CWEs simultaneously achieves 102 TPs and 137 FNs (see Tables 9 and 10), whereas the combined totals from four individually trained CT5 models, one per CWE, yield only 85 TPs and 154 FNs (see Tables 6 and 7) in total. For clarity, 85 TPs come from the sum of individually trained models ($37 + 28 + 6 + 14$), where each model is trained on $D_{i,train}$ and tested on $D_{i,test}$ for $i = \{119, 125, 787, 20\}$. This pattern generally persists across other M_n (except where $n = 3$) for CT5, showing its ability to detect more TPs with varying efficiency.

However, the trend is not consistent for other models. For CB and UC, M_n often yield mixed results compared to their $\sum M_i$. While M_n occasionally improves performance for UC, the improvements are inconsistent. CB, on the other hand, frequently performs better in $\sum M_i$ than M_n. Therefore, the observed advantage of cumulative training primarily holds true for CT5, whereas CB and UC show more varied behavior across M_n. The fact that CB and UC are encoder-based and CT5 is encoder-decoder-based might play an important role.

Additionally, we see a similar pattern in the comparison within different M_n. The benefits of cumulative training diminish for both encoder-based architectures

as more CWEs are added. While M_n initially shows gains in TP, these gains become inconsistent. For instance, as shown in Table 10, UC trained with $D^{cum}_{train,8}$ achieves the highest TP count of 157. But, when trained with $D^{cum}_{train,10}$, it sees a decline with TP reduced to 139. The same goes with CB on several occasions. This trend likely occurs because the model begins to encounter overlapping patterns among the CWEs. Once the model's representational capacity is saturated, introducing additional CWEs does not yield substantially new information. Instead, it may lead to performance deterioration when the model attempts to generalize across CWEs outside its learned feature cluster, forcing it to accommodate a broader and potentially conflicting feature space.

While cumulative training generally enhances classification through increased feature representation, this improvement diminishes as less representative CWEs are added, leading to a consistent increase in FN across all model types (Tables 9 and 10). This observation may also explain why our model's F1 score outperformed PrimeVul [Din+25], as discussed in Section 4.3. Unlike PrimeVul, which includes all available datasets from all CWEs in its training set, our model focuses exclusively on the top 10 CWEs, avoiding the negative impact of training on sparse or inconsistent data, thereby specializing in well-represented CWE features within the learned features cluster.

The TP' metric further illustrates that cumulative models can identify vulnerabilities in CWEs they were not explicitly trained on. This demonstrates the inherent transferability of vulnerability features between different CWEs. Consequently, cumulative models not only improve detection within their trained domains but also show the ability to generalize beyond them.

The $F1'$ metric initially rises as more CWEs are integrated into the training process, indicating enhanced overall detection capabilities. Eventually, however, $F1'$ tends to plateau, reflecting a saturation point in the model's ability to incorporate additional CWE patterns into a more generalized detection framework. This plateauing could also be attributed to the limited availability of representative datasets for certain CWEs, which constrains the model's capacity to learn distinguishing features effectively.

The observed transferability between CWEs indicates potential benefits in identifying and training on clusters of related vulnerabilities. By carefully selecting CWEs with strong transferability into one class, future efforts could refine training methodologies to achieve the best performance.

RQ2 Summary. Our RQ2 analysis shows that as more common vulnerability types are incrementally included in the training set, models initially detect more true positives, suggesting greater versatility. However, these improvements are not limitless. After a certain point, performance gains plateau or decline, indicating that too many vulnerability types can overwhelm the model, diminishing its per-type recall. Thus, while expanding coverage may enhance generalization and precision, it also reduces the model's specialization and accuracy for individual vulnerabilities, revealing a fundamental trade-off between broad coverage and

per-type detection quality. Although F1 is limited, it remains the only common ground for comparison. On this metric, our cumulative models (M_n) outperform PrimeVul [Din+25], demonstrating that training on multiple well-chosen CWEs can improve upon established baselines.

5 Threats to Validity

While our study provides valuable insights into the performance of language models for vulnerability detection across different CWEs, there are a few potential threats to the validity of our findings that should be acknowledged.

First, our analysis focuses on the top 10 most prevalent CWEs. Although these CWEs represent a significant portion of real-world vulnerabilities, they do not exhaustively cover all possible vulnerability types. It is possible that the performance trends and observations we noted may not fully generalize to less common CWEs or to the entire spectrum of vulnerabilities. Future work could expand the scope of the study to include a broader range of CWEs and assess whether the patterns we identified hold consistently.

Second, due to the limitations of the PrimeVul-v0.1 dataset, which does not include timestamp information, we were unable to split the data based on chronological order. This introduces a potential "time travel" issue, where the models may be trained on vulnerability patterns that emerged after some of the testing samples were created (temporal leakage [AIP24]). In an ideal scenario, the training data should precede the testing data chronologically to simulate a realistic deployment setting. The absence of time-based splitting may lead to overly optimistic performance estimates. Future research could strive to curate datasets with clear chronological information to enable more rigorous, time-aware evaluations of vulnerability detection models.

Despite these limitations, we believe our work takes a meaningful step toward understanding the nuances of language model performance across different vulnerability types and the implications of incrementally expanding the scope of vulnerability detection. Our findings highlight the trade-offs between specialization and generalization and emphasize the challenges in achieving both high recall and precision as the diversity of vulnerabilities increases.

6 Conclusion and Future Works

In this study, we re-examined the vulnerability detection task from a viewpoint that differs from the current *de-facto* binary classification approach. Instead of treating all vulnerability types as a single homogenous label, we narrowed the focus to classifying vulnerabilities by their specific CWE types. Using the PrimeVul dataset, we investigated how code LMs perform when answering more granular questions like, "Is this C/C++ code vulnerable to a specific CWE type?"

Key Findings. Our results show that code LMs trained on a single or a small subset of CWE types achieve higher recall for those targeted vulnerabilities compared to models trained on a broader range of vulnerability types. However, as the scope of CWE types broadens, complexity increases, and per-type recall often declines. In particular, we show that it is challenging to correctly identify a specific class of vulnerability in a dataset containing all types of vulnerabilities, as evidenced by significant drops in precision when models are evaluated on comprehensive test sets. While including more CWEs can improve some aggregate metrics and reduce certain types of false positives, this generally comes at the cost of reduced detection acuity for individual vulnerabilities. This trade-off stresses a tension between specialization and generalization: models optimized for a particular vulnerability type excel at that task but fare worse when facing a diverse set of vulnerabilities.

Evaluating models solely with standard classification metrics like F1 can be misleading in this multi-CWE context. Due to data imbalance and the complexity of covering multiple vulnerability types, F1 often fluctuates based on precision rather than meaningful improvements in recall. To gain a deeper understanding, we expanded our evaluation scope by testing models on broader datasets that include CWEs outside their original training targets. In these extended evaluations, we examined standard metrics such as true positives (TP') and F1-score ($F1'$) in a broader context. While TP' and $F1'$ are not new metrics, this different evaluation scenario allowed us to observe how models trained on one CWE might inadvertently detect vulnerabilities associated with other CWEs. This shed light on the transferability of learned representations and revealed that certain CWEs (i.e., CWE-119, CWE-20) have more universally detectable patterns, while others (i.e., CWE-190, CWE-703) are more unique and less likely to enhance cross-CWE generalization. Moreover, by focusing on the top 10 CWEs, our models outperform the reported PrimeVul baseline in F1, showing that careful selection of CWEs can lead to stronger aggregate metrics.

When models were trained cumulatively on multiple CWEs, they often identified more vulnerabilities overall, initially improving both true positives and reducing false negatives. However, these gains tended to plateau as more CWEs were added, suggesting a representational saturation point. The interplay between specialization and generalization shows the need for strategic CWE selection and clustering. By training on representative groups of CWEs that share overlapping traits, future research could enhance both coverage and efficiency.

Implications. The outcomes suggest that current benchmarks, where all vulnerability types are lumped into a single label, do not fully capture the complexities of real-world vulnerability detection in C/C++. Practical tools may need hybrid strategies: specialized models for high-priority CWEs supplemented by broader models capable of screening a wide range of vulnerabilities.

Future works. Several avenues exist for future research. Expanding the analysis beyond the top 10 CWE types would confirm whether our observed patterns

hold for less frequent and more diverse vulnerabilities. This includes identifying CWE clusters with overlapping features that could guide more efficient training strategies, i.e., finding the optimum subset of CWEs that yields better generalization while minimizing training complexity. Additionally, incorporating chronological splits and real-world temporal data would lead to more realistic performance estimates, mitigating the risk of "time travel" biases. Beyond data considerations, future work could explore model architectures or training techniques that dynamically balance generalization and specialization, i.e., via multi-task learning [CZY24] or hierarchical classification [Wan+23] strategies. While our focus has been on C/C++ code, understanding whether similar trade-offs arise in other programming languages would further enrich the field's collective insights.

Acknowledgements

This work is partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation and by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research. The computations and data handling were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

References

[AIP24] A. Apicella, F. Isgrò, and R. Prevete. *Don't Push the Button! Exploring Data Leakage Risks in Machine Learning and Transfer Learning*. 2024. arXiv: 2401.13796 [cs.LG].

[Bal99c] T. Ball. "The concept of dynamic analysis." In: *ACM SIGSOFT Software Engineering Notes* 24.6 (1999), pp. 216–234.

[BNM21b] G. Bhandari, A. Naseer, and L. Moonen. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software." In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 30–39.

[Bro+20] T. Brown et al. "Language Models are Few-Shot Learners." In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. New York, NY, USA: Curran Associates, Inc., 2020, pp. 1877–1901.

[Cha+22a] S. Chakraborty et al. "Deep Learning Based Vulnerability Detection: Are We There Yet?" In: *IEEE Transactions on Software Engineering* 48.09 (Sept. 2022), pp. 3280–3296.

[Cha+22c] S. Chakraborty et al. "NatGen: generative pre-training by "naturalizing" source code." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 18–30.

[Che+23b] Y. Chen et al. "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection." In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 654–668.

[Cho+14] K. Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation." In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by A. Moschitti, B. Pang, and W. Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[CM04] B. Chess and G. McGraw. "Static analysis for security." In: *IEEE Security & Privacy* 2.6 (2004), pp. 76–79.

[CV95] C. Cortes and V. Vapnik. "Support-vector networks." In: *Machine learning* 20 (1995), pp. 273–297.

[CZY24] S. Chen, Y. Zhang, and Q. Yang. "Multi-Task Learning in Natural Language Processing: An Overview." In: *ACM Comput. Surv.* 56.12 (July 2024).

[Dev+19b] J. Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186.

[Din+25] Y. Ding et al. "Vulnerability Detection with Code Language Models: How Far Are We?" In: *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 469–481.

[Fan+20] J. Fan et al. “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries.” In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 508–512.

[Fen+20b] Z. Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by T. Cohn, Y. He, and Y. Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[Guo+22] D. Guo et al. “UniXcoder: Unified Cross-Modal Pre-training for Code Representation.” In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by S. Muresan, P. Nakov, and A. Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225.

[Guo+24a] D. Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE].

[HS97] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory.” In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[JW23] M. A. Jette and T. Wickberg. “Architecture of the Slurm Workload Manager.” In: *Job Scheduling Strategies for Parallel Processing: 26th Workshop, JSSPP 2023, St. Petersburg, FL, USA, May 19, 2023, Revised Selected Papers*. St. Petersburg, FL, USA: Springer-Verlag, 2023, pp. 3–23.

[Le+22] H. Le et al. “CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning.” In: *NeurIPS*. 2022.

[Li+18b] Z. Li et al. “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection.” In: *NDSS*. The Internet Society, 2018.

[Li+22] Z. Li et al. “SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities.” In: *IEEE Transactions on Dependable and Secure Computing* 19.4 (2022), pp. 2244–2258.

[Li+23] R. Li et al. “StarCoder: may the source be with you!” In: *Transactions on Machine Learning Research* (2023). Reproducibility Certification.

[Lin+20] G. Lin et al. “Software Vulnerability Detection Using Deep Neural Networks: A Survey.” In: *Proceedings of the IEEE* 108.10 (2020), pp. 1825–1848.

[Liu+20] Y. Liu et al. *Ro{BERT}a: A Robustly Optimized {BERT} Pretraining Approach*. 2020.

[Mir+23] Y. Mirsky et al. "VulChecker: Graph-based Vulnerability Localization in Source Code." In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6557–6574.

[Neu+07] S. Neuhaus et al. "Predicting vulnerable software components." In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 529–540.

[Nij+23] E. Nijkamp et al. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis." In: *The Eleventh International Conference on Learning Representations*. 2023.

[Nik+21a] G. Nikitopoulos et al. "CrossVul: a cross-language vulnerability dataset with commit data." In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1565–1569.

[Rad+19] A. Radford et al. "Language models are unsupervised multitask learners." In: *OpenAI blog* 1.8 (2019), p. 9.

[Raf+20] C. Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer." In: *J. Mach. Learn. Res.* 21.1 (Jan. 2020).

[Sca+09] F. Scarselli et al. "The Graph Neural Network Model." In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80.

[Tha+22] C. Thapa et al. "Transformer-Based Language Models for Software Vulnerability Detection." In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 481–496.

[Vas+17] A. Vaswani et al. "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. New York, NY, USA: Curran Associates, Inc., 2017.

[Wan+21c] Y. Wang et al. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens et al. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708.

[Wan+23] Y. Wang et al. “Towards Better Hierarchical Text Classification with Data Generation.” In: *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by A. Rogers, J. Boyd-Graber, and N. Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 7722–7739.

[Wei+24] Y. Wei et al. “Magicoder: Empowering Code Generation with OSS-Instruct.” In: *Forty-first International Conference on Machine Learning*. 2024.

[Yam+14] F. Yamaguchi et al. “Modeling and Discovering Vulnerabilities with Code Property Graphs.” In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 590–604.

[Zhe+24] T. Zheng et al. “OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement.” In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 12834–12859.

[Zho+19b] Y. Zhou et al. “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.” In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. New York, NY, USA: Curran Associates, Inc., 2019.

[Zho+24] X. Zhou et al. “Comparison of static application security testing tools and large language models for repo-level vulnerability detection.” In: *arXiv preprint arXiv:2407.16235* (2024).

Vulnerability Detection in Popular Programming Languages with Language Models

Abstract

Vulnerability detection is crucial for maintaining software security, and recent research has explored the use of Language Models (LMs) for this task. While LMs have shown promising results, their performance has been inconsistent across datasets, particularly when generalizing to unseen code. Moreover, most studies have focused on the C/C++ programming language, with limited attention given to other popular languages. This paper addresses this gap by investigating the effectiveness of LMs for vulnerability detection in JavaScript, Java, Python, PHP, and Go, in addition to C/C++ for comparison. We utilize the CVEFixes dataset to create a diverse collection of language-specific vulnerabilities and preprocess the data to ensure quality and integrity. We fine-tune and evaluate state-of-the-art LMs across the selected languages and find that the performance of vulnerability detection varies significantly. JavaScript exhibits the best performance, with considerably better and more practical detection capabilities compared to C/C++. We also examine the relationship between code complexity and detection performance across the six languages and find only a weak correlation between code complexity metrics and the models' F1 scores.

Syafiq Al Atiiq, Christian Gehrmann, Kevin Dahlén. "Vulnerability Detection in Popular Programming Languages with Language Models". Submitted to 20^{th} *International Conference on Availability, Reliability and Security, ARES 2025, Ghent, Belgium*.

1 Introduction

Vulnerability detection is a critical component of software security, as undetected vulnerabilities can lead to severe consequences [McG04]. With the increasing complexity of software, manual detection becomes impractical, requiring the development of automated techniques[1]. In recent years, deep learning-based approaches have shown promise in vulnerability detection, particularly using language model (LM) [Li+18a]. LMs, such as BERT [Dev+19a], GPT [Rad+19], and Sonnet [Tem+24], have demonstrated outstanding performance in code understanding and generation tasks. These models can capture rich semantic and syntactic information from a large codebase, potentially well-suited for analyzing source code [Alo+19].

Several studies have explored LM's application for vulnerability detection in C/C++ code [Cha+22b; Zho+19a; Ull+24]. However, the results have been mixed, with challenges like high false positive rates (FPR) and limited generalization to unseen code [Che+23a; Din+25; Ati+24]. This raises a question about the effectiveness of LM for real-world scenarios. While C/C++ is a widely used language, particularly in system-level software, there is a growing need to address the same problem in other programming languages[2]. Languages such as JavaScript, Java, Python, PHP, and Go are extensively used in web development, enterprise applications, and data analysis[3]. Vulnerabilities in these languages may have significant implications, given their widespread adoption and the sensitive nature of the applications they support[4].

To bridge this gap, this paper investigates the effectiveness of LMs for vulnerability detection in popular programming languages. Our experimental evaluation covers JavaScript, Java, Python, PHP, and Go. We use the CVEFixes dataset [BNM21a], containing a diverse set of vulnerabilities across multiple languages, and preprocess it to create language-specific subsets. These languages are chosen because they have the highest number of samples in the dataset. By fine-tuning and evaluating state-of-the-art (SOTA) language models on these subsets, we assess their performance in detecting vulnerabilities across different languages and address the following research questions:

RQ1. How does the detection performance vary for different programming languages when using SOTA LMs on code from the same curated dataset?
Result. Our findings indicate that the detection performance of finetuned LM varies significantly across different languages. Specifically, language like JavaScript demonstrated higher vulnerability detection performance, achieving better F1 scores, suggesting a more effective balance between pre-

[1] https://lcamtuf.coredump.cx/afl/
[2] `https://www.tiobe.com/tiobe-index/`
[3] `https://survey.stackoverflow.co/2024/`
[4] `https://owasp.org/www-project-top-ten/`

cision and recall. In contrast, C/C++ showed lower performance, with significantly lower F1 scores.

RQ2. Is there any dependency between code complexity and the detection performances using LM?
Result. Our analysis did not find a strong correlation between code complexity and the detection capabilities of LM. We computed various code complexity metrics, including Cyclomatic Complexity, Halstead Effort, Token Length, Halstead Volume, Halstead Difficulty, and Number of Lines of Code (NLOC), and assessed their relationship with the model's F1 scores across different languages. The correlation coefficients were weak and not statistically significant, indicating that code complexity, as measured by these metrics, may not be a determining factor in the effectiveness of LM-based vulnerability detection. This suggests that other factors, such as language-specific characteristics, dataset quality, and the inherent ability of the models to generalize across different coding styles, may play a more substantial role in influencing detection performance.

Open Science. In order to contribute to the open science community, we have made the dataset, scripts (including those for data processing, model training, and model evaluation), and experimental results from our study publicly available on GitHub at the following link: `https://github.com/syafiq/llm_vd`.

Paper Organization. The remainder of this paper is organized as follows. Section 2 presents the background and related work on vulnerability detection techniques and language models. Section 3 provides the problem definition of our work. Section 4 describes our dataset, models, and our fine-tuning setup. Section 5 presents the results and analysis of our experiments. Section 6 discusses the limitations of our work, followed by conclusions and the implications of our findings in section 7.

2 Background and Related Work

2.1 Vulnerability Detection Techniques

Over the years, various techniques have been developed to detect vulnerabilities in software systems, ranging from traditional approaches to more advanced deep learning-based methods [Lin+20; Lin+24; Lia+25].

Traditional Approaches

Traditional vulnerability detection approaches include manual code review [Sad+18b], static analysis [Tho21], and dynamic analysis [Bal99b]. Manual code review is a labor-intensive process where security experts manually inspect the source code to identify vulnerabilities [BB22]. Static analysis techniques examine

the source code without executing it to identify potential vulnerabilities [CM04]. Dynamic analysis involves executing the program with specific inputs and monitoring its behavior to detect runtime vulnerabilities. [SAB10]. While these traditional approaches have effectively detected certain vulnerabilities, they have limitations. Manual code review is time-consuming [LE20] and relies heavily on the expertise of the reviewers. Static analysis techniques can suffer from high false positives [Tym17], while dynamic analysis may miss vulnerabilities not triggered by the specific inputs used during testing.

Non-Language Model-based Deep Learning Approaches

In recent years, deep learning-based approaches have gained attention for their ability to automatically learn features from large amounts of data and detect complex patterns. These approaches have been applied to various software engineering tasks, including vulnerability detection [Cha+22b; Li+18a; Zho+19a]. Several studies have explored using deep learning models, such as convolutional neural networks (CNNs) and graph neural networks (GNNs). VulDeePecker [Li+18a] uses CNNs to detect vulnerabilities in C/C++ code. Devign [Zho+19a] combines GNNs with code property graphs to detect vulnerabilities in C/C++ programs.

While the deep learning-based approaches have shown promising results, they often require extensive feature engineering and may struggle to capture the complex semantics and dependencies present in source code [Yan+22]. Traditional deep learning models typically rely on manual feature engineering to extract relevant information from the code, which can be time-consuming and require domain expertise. Moreover, these models may have difficulty understanding the intricate relationships between code elements, such as variable scoping, control flow, and data dependencies, and capturing the context and intent behind the code. Attention mechanisms used in transformer-based models have been proposed to address these limitations. These mechanisms can help capture long-range dependencies and global context. However, even with these advancements, deep learning models may still struggle to fully capture source code's complex semantics and nuances, especially when dealing with diverse programming languages and coding styles.

2.2 Vulnerability Detection with Language Model

The language model (LM) is a class of deep learning models trained on vast amounts of text data to learn rich natural language representations. These models, such as BERT [Dev+19a], GPT [Rad+19], Gemini [Tea24], and Deepseek [Dai+24], have achieved state-of-the-art performance on a wide range of natural language tasks. LMs are typically based on transformers [Vas+17], which use self-attention mechanisms to capture long-range dependencies in the input. By pre-training on large corpora of text data, LM learns to understand the structure

and semantics of natural language, enabling them to generate coherent and contextually relevant text. The success of LM in natural language has sparked interest in applying them to other domains, such as source code analysis and generation. Several studies have explored the use of LM for tasks such as code completion [Svy+20], code summarization [Fen+20a], and bug detection [Li+24].

LM for Code Analysis

Source code, like natural language, shows rich structural and semantic properties. LM, with their ability to capture complex patterns, have shown promise in analyzing source code [Alo+19; Nam+24; Shi+24]. Several LMs have been specifically designed for code analysis. CodeBERT [Fen+20a] is a pre-trained model that learns from a large corpus of programming language data, enabling it to understand the semantics of code. GraphCodeBERT [Guo+21] extends CodeBERT by incorporating graph-based representations of code, capturing the structural information present in abstract syntax trees and data flow graphs.

Vulnerability Detection using LM

The application of LM for vulnerability detection has gained attention in recent years, mainly in the context of C/C++ code. Several studies investigated the effectiveness of LM in detecting vulnerabilities and reported promising results and significant challenges.

In C/C++, two notable datasets have been introduced. DiverseVul [Che+23a] is a dataset containing 18,945 vulnerable functions spanning 150 CWEs and 330,492 non-vulnerable functions. The authors study 11 model architectures from 4 families (GNN [Sca+09], RoBERTa [Liu+20], GPT-2 [Rad+19], and T5 [Wan+21c]) and find that LM outperforms GNN, especially when trained on larger datasets. However, the model still struggles with high FPR and limited generalization. As a follow-up, PrimeVul [Din+25] is another C/C++ dataset designed to address the limitations of existing datasets, i.e., poor data quality, low label accuracy, and high duplication rates. The authors evaluate various LMs on PrimeVul and find that their performances are significantly lower than reported for the previous datasets. The performance remains poor even with advanced training techniques and larger models.

The major issue with the detection results reported from the PrimeVul [Din+25] and DiverseVul [Che+23a] is is the low F1 scores. PrimeVul highlights the significant underperformance of LM when faced with realistic, diverse, and challenging vulnerabilities, even with advanced training techniques and SOTA models. Similarly, the DiverseVul study pinpoints a significant challenge for models to generalize to unknown test projects on the vulnerability detection task.

Furthermore, the vulnerability types diversity, each with its own unique characteristics, code semantics, and patterns, makes it challenging for a single binary classifier to learn and detect vulnerabilities across all categories effectively [Ati+24].

The imbalance between vulnerable and non-vulnerable code samples in datasets can also lead to biased models that struggle to learn and detect vulnerabilities effectively [Che+23a; Din+25]. Inadequate training techniques have also been identified as a limitation. Advanced training techniques, such as class weights and contrastive learning (both performed in [Che+23a; Din+25]), have shown limited success in improving the performance. Fundamentally, new approaches may be needed to address this.

PrimeVul highlights key areas where the current LM falls short. First, prior work focuses on function-level analysis without considering the broader context, such as interprocedural data flows, making detecting vulnerabilities challenging even for humans. Second, LM makes decisions primarily based on textual similarity without considering the underlying root causes. Finally, posing vulnerability detection as a binary classification problem might be oversimplistic. A more nuanced approach that decomposes the problem into sub-problems and teaches the model to reason about each step might be more effective.

3 Problem Definition

As shown in our review of previous results in Section 2, while LM looks promising in vulnerability detection for C/C++ code, significant challenges remain, including low F1 scores, limited generalization, vulnerability types diversity, data imbalance, and inadequate training techniques. The models built for C/C++ did not work as expected, spurring the need to explore whether similar problems appear for other programming languages. Despite this fact, few efforts have been devoted to investigating if similar issues exist for other programming languages or even if there are large detection performance differences between languages. We address this research gap by making an experimental evaluation of the differences in vulnerability detection using LMs for JavaScript, Java, Python, PHP, and Go. We also include figures with C/C++ to make a proper comparison between our results and the previous results on C/C++ possible. Proper comparison can only be done if a reasonably similar dataset for all languages is available. To address this issue, we present a cleaned new dataset that can be used to make sound comparisons. With the help of such a dataset, the related research question also arises: "If we have performance differences between different languages, can this difference be due to code complexity variations?". We address this research question as well, presenting figures on the correlation between vulnerability detection performance and code complexity for the given dataset.

4 Dataset, Models and Setup

4.1 Dataset

Several datasets have been proposed for building vulnerability detection systems, including: (i) C/C++ based datasets such as DiverseVul [Che+23a] and PrimeVul [Din+25]; (ii) project-specific datasets like ReVeal [Cha+22b], which focuses on the Linux Debian kernel and Chromium projects; and (iii) cross-language datasets such as CrossVul [Nik+21b] and CVEFixes [BNM21a]. These datasets provide diverse vulnerabilities and have been widely used in the research community. Among these, only CrossVul and CVEFixes are suitable for our purpose. Both CrossVul and CVEFixes extract the data based on the same Common Vulnerabilities and Exposures (CVE) records in the public U.S. National Vulnerability Database (NVD)[5], so using both would create unnecessary duplication. However, only CVEFixes provides the data update regularly (as well as an open-source script[6] to scrape the data on our own if the newest data is unavailable yet). After considering the requirements of our study, including the need for cross-language vulnerability data and regular updates, we ultimately selected CVEFixes as the most suitable dataset to serve as the foundation of our work.

CVEfixes initial release spans all published CVE records up to June 9, 2021, covering 5365 CVE records for 1754 open-source projects. 5495 vulnerability fixing commits are obtained from the projects' version control systems and linked to information from the corresponding CVE records, such as CVE-IDs, reference links, severity scores, CWE type, and other descriptive information. The latest release covers all published CVEs up to 23 July 2024. Figure 1 shows the distribution of programming languages in the CVEFixes dataset based on the latest scraped data from 2024. The dataset consists of 277,948 entries, with 126,599 classified as vulnerable and 151,349 as non-vulnerable. It covers many languages, with JavaScript being the most prevalent, followed by PHP, Java, Python, and Go. These top 5 languages (excluding C, C++, and unknown) were selected as the focus of our study due to their significant representation.

To validate the generalizability of our findings, we tested our models on independent datasets where available. For Java, we used the MegaVul dataset [Ni+24] (only the Java part[7]). For Python, we tested on the synth-vuln-fixes dataset[8], a curated dataset of Python vulnerabilities generated by GPT-4 and validated through both human review and static analysis. For PHP, we used the PHP vulnerability test suite from SARD[9] (Software Assurance Reference Dataset) [SF16]. It's important to note that the SARD PHP test cases are synthetic, meaning they were specifically created as examples with well-characterized weaknesses. We were un-

[5] https://nvd.nist.gov/

[6] https://github.com/secureIT-project/CVEfixes

[7] https://github.com/Icyrockton/MegaVul

[8] https://huggingface.co/datasets/patched-codes/synth-vuln-fixes

[9] https://samate.nist.gov/SARD/test-suites/103

able to find suitable alternative datasets containing vulnerabilities and their fixes for JavaScript and Go. For languages where we had alternative datasets, this cross-dataset validation helps assess how well our models generalize beyond the CVEFixes data.

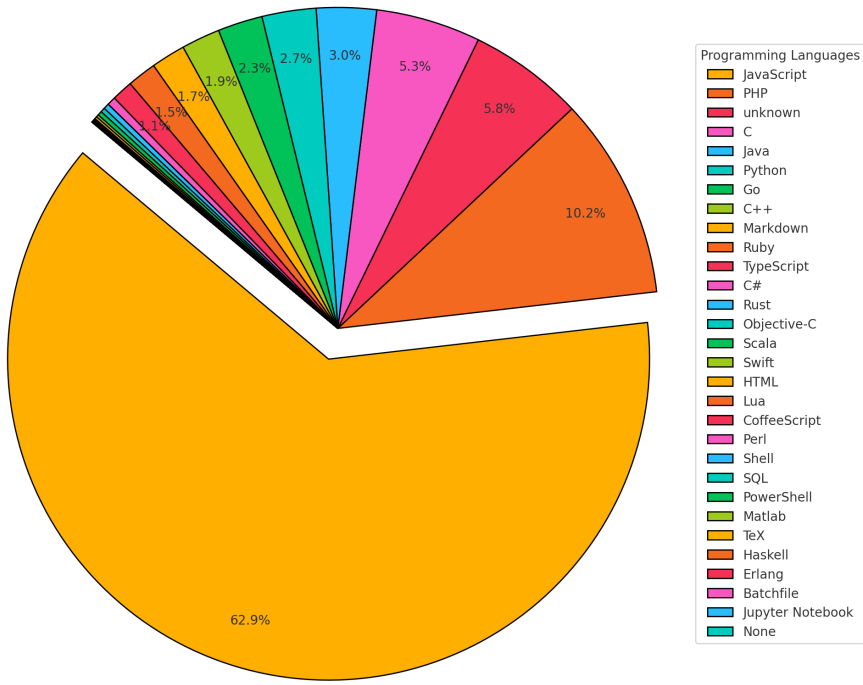

Figure 1: Distribution of programming languages in the CVEfixes dataset.

Data Preprocessing and Filtering

We first split CVEfixes based on the programming language. For each subset, we perform the following preprocessing steps:

1. Duplication removal: We identify and remove any duplicate entries in the dataset to avoid data leakage [Ros+24].

2. Train-test splitting: To maintain temporal integrity and simulate a realistic evaluation scenario, we divide the data into training and testing sets based on the commit timestamps using stratified sampling. Stratified sampling is a technique that ensures each class (in this case, vulnerable and non-vulnerable) is proportionally represented in both the training and testing sets. This is particularly important when dealing with imbalanced datasets, as it helps maintain the original class distribution and prevents bias towards the majority class. After stratifying the data, we split it so that the model is trained on data from earlier timestamps and evaluated on data from later timestamps. This method ensures that the model's performance is assessed on future, unseen vulnerabilities, thereby avoiding any inadvertent leakage of future information into the training process. This also tests the model's ability to catch vulnerabilities that have not been seen before.

Table 1 shows the distribution of the five subsets (and C/C++ for comparison) used in this paper.

Language	Vulnerable	Non-vulnerable	Total
JavaScript	46,802 (82,999)	53,198 (91,929)	100,000 (174,928)
PHP	4,758	23,499	28,257
C/C++	8,299	11,761	20,060
Java	3,102	5,285	8,387
Python	2,775	4,793	7,568
Go	1,880	4,403	6,283

Table 1: Vulnerable and non-vulnerable counts by programming language. Note: for JavaScript, we only used the first 100,000 entries (57.17% of the total data).

4.2 Models

We investigate the performance of the language-specific subset data for the vulnerability detection task with four LMs. Table 2 presents the four LMs evaluated in this study, along with their parameter sizes and architectures. DeepSeek-Coder, with 1.3 billion parameters, is the largest model among the four, followed by CodeBERT and UnixCoder, both having 125 million parameters, and finally, CodeT5,

Model	Acronym	Parameters	Arch	Method
CodeT5 [Wan+21c]	CT5	60 M	Enc-Dec	Fine-tune
CodeBERT [Fen+20a]	CB	125 M	Encoder	Fine-tune
UnixCoder [Guo+22]	UC	125 M	Encoder	Fine-tune
DeepSeek-Coder [Guo+24b]	DS-C	1.3 B	Decoder	Fine-tune

Table 2: LMs we study.

with 60 million parameters. We perform full fine-tuning on all the models and evaluate them independently.

4.3 Experimental Setup

We design the experiments with all the LMs following the existing benchmarks [Che+23a; Din+25; Lu+21a]. This includes setting the learning rate to be 2×10^{-5}, running the fine-tuning for ten epochs, and a batch size of one per GPU. We employ the AdamW optimizer and use gradient accumulation with a step size of 8 to simulate a larger effective batch size. The learning rate is scheduled using a linear warmup for 1000 steps followed by a linear decay. The LMs are fine-tuned using the PyTorch deep learning framework on a node of NVIDIA A100 GPUs.

5 Results and Analysis

We aim to evaluate the performance of LMs in detecting vulnerabilities across various popular programming languages. This is particularly intriguing given that recent research findings on C/C++ indicate significant challenges related to detection performance in real-world applications, as discussed in Section 1.

5.1 Evaluation Metrics

We employ several evaluation metrics to assess the performance:

- **Accuracy**: The proportion of correctly classified samples (both vulnerable and non-vulnerable) out of the total number of samples.
- **Precision**: The proportion of correctly identified vulnerable samples out of all samples classified as vulnerable by the model.
- **Recall**: The proportion of correctly identified vulnerable samples out of all actual vulnerable samples in the dataset.
- **F1 Score**: The harmonic mean of precision and recall, providing a balanced measure of the model's performance.
- **False Positive Rate (FPR)**: The proportion of non-vulnerable samples incorrectly classified as vulnerable out of all non-vulnerable samples.

5.2 RQ1. Results regarding language-dependent detection performance

The analysis of vulnerability detection performance from Table 3 in models trained on non-C/C++ languages (JavaScript, PHP, Java, Python, and Go) reveals varying levels of effectiveness. These models achieve accuracy ranging from 50% to 93%

Language	Model	Train	Test	Acc ↑	F1 ↑	Prec ↑	Rec ↑	FPR ↓
JavaScript[1]	CT5	CVEFixes	CVEFixes	50.53	27.17	50.80	18.54	17.79
	CB			71.46	69.25	74.62	64.60	21.75
	UC			70.11	70.23	69.61	70.86	30.64
	DS-C			69.59	65.55	75.13	58.14	19.07
PHP[1]	CT5	CVEFixes	CVEFixes	82.30	24.32	27.10	22.05	8.78
	CB			81.62	28.84	28.79	28.90	10.58
	UC			81.67	38.08	33.72	43.73	12.72
	DS-C			82.84	36.59	34.95	38.40	10.58
	CT5		SARD	47.80	46.41	33.78	74.10	63.74
	CB			38.50	39.41	28.17	65.57	73.38
	UC			53.10	17.86	19.17	16.72	30.93
	DS-C			37.80	25.42	20.04	34.75	60.86
Java[1]	CT5	CVEFixes	CVEFixes	53.98	62.75	52.30	78.41	69.90
	CB			57.22	65.54	54.45	82.30	67.30
	UC			57.83	65.47	54.99	80.88	64.71
	DS-C			68.59	69.18	67.17	71.33	34.08
	CT5		MegaVul	59.75	66.56	56.93	80.10	60.60
	CB			58.80	64.27	56.74	74.10	56.50
	UC			65.65	70.11	62.04	80.60	49.30
	DS-C			57.20	63.88	55.26	75.70	61.30
Python[1]	CT5	CVEFixes	CVEFixes	54.89	45.57	41.87	50.00	42.14
	CB			59.24	47.74	46.28	49.28	34.72
	UC			55.03	52.24	43.61	65.11	51.09
	DS-C			60.19	53.27	47.85	60.07	39.74
	CT5		synth-vuln-fixes	55.47	55.58	55.45	55.72	44.78
	CB			22.14	0.63	0.87	0.49	56.22
	UC			35.82	40.55	37.77	43.78	72.13
	DS-C			41.04	1.66	5.00	0.99	18.91
Go[1]	CT5	CVEFixes	CVEFixes	89.19	37.25	29.23	51.35	8.29
	CB			86.66	43.17	29.41	81.08	12.97
	UC			92.57	55.10	44.26	72.97	6.12
	DS-C			85.30	41.61	27.68	83.78	14.59
C/C++[1]	CT5	CVEFixes	CVEFixes	57.13	33.06	45.25	26.05	21.58
	CB			57.59	42.71	47.37	38.88	29.59
	UC			59.50	1.25	71.43	0.63	0.17
	DS-C			60.02	34.96	51.61	26.43	16.97
	CT5		PrimeVul	71.28	3.86	2.09	26.05	27.69
	CB			48.51	3.11	1.62	37.34	51.23
	UC			96.97	1.57	2.79	1.09	0.86
	DS-C			65.55	2.02	1.08	16.03	33.33
C/C++[2]	CT5	DiverseVul	DiverseVul	94.91	9.39	13.35	7.24	1.78
	CB			94.19	11.94	13.34	10.80	2.65
C/C++[2]	CT5	PrimeVul	PrimeVul	96.67	19.7	-	-	-
	CB			96.87	20.86	-	-	-
	UC			96.86	21.43	-	-	-
C/C++[2]	DS-C	Gen2Spec	Gen2Spec	75.68	23.36	13.76	77.35	24.41
			PrimeVul	76.46	14.58	7.93	90.71	23.86

Table 3: The performance of LMs (using acronyms from table 2) to the respective programming languages.

[1] Measured by us.

[2] Taken from the respective paper with the size of the datasets can be seen in table 4. (-) means no data.

Dataset	Vulnerable	Non-Vulnerable	Total
DiverseVul [Che+23a]	18,945	311,997	330,492
PrimeVul [Din+25]	6,968	228,800	235,768
Gen2Spec [Ati+24]	16,955	295,587	312,542
C/C++@CVEFixes	8,299	11,761	20,060

Table 4: Size of previous C/C++ datasets.

and F1 scores spanning from 24% to 70%, indicating moderate to good performance in identifying vulnerabilities in these languages. However, the FPRs for non-C/C++ languages are often higher, ranging from 6% to 69%, depending on the language and dataset used. The higher FPR suggests that while these models are generally capable of detecting vulnerabilities, they also tend to incorrectly flag more code snippets as vulnerable, which could lead to unnecessary manual review efforts.

The higher FPR scores observed can be attributed to several factors. While the CVEfixes dataset used in our study covers a significant portion (80%) of the available vulnerability fixes from the NVD database, the dataset sizes for individual languages like Go, Java, and Python are relatively smaller compared to the C/C++ datasets used in previous studies. However, it is important to note that these smaller dataset sizes are a result of the inherent distribution of vulnerabilities across languages in the NVD database (reported real-world representation) and not a limitation of the CVEfixes dataset itself.

In addition to dataset size, other factors such as dataset quality, diversity, and the inherent characteristics of the programming languages may also contribute to the observed differences in performance. The relatively smaller datasets for languages like Go, Java, and Python in the current study may impact the models' ability to learn and generalize effectively, leading to higher FPR scores despite their overall better performance in terms of F1 score compared to the results obtained in DiversVul and PrimeVul for C/C++. Even if this is the case, the datasets used in our comparison are rather similar, allowing a better comparison between the models under the same conditions. This indicates a better ability to detect vulnerabilities with LM for JavaScript compared to C/C++.

Among the non-C/C++ languages, the performance varies. JavaScript, Java, and Python perform better overall than PHP and Go. Models trained on JavaScript, Java, and Python achieve moderate accuracy, ranging from 50% to 71%, and F1 scores between 27% and 70%. These results suggest that the models are relatively effective in identifying vulnerabilities in these languages, although there is still room for improvement.

For PHP, the high accuracy (81% to 83%) and low F1 scores (24% to 38%) could result from the significant class imbalance in the dataset, with around 4,758 vulnerable samples and 23,499 non-vulnerable samples. Despite the stratified train-test splitting approach, the limited number of vulnerable samples in the

training set might hinder the models' ability to learn effective patterns for detecting vulnerabilities, leading to a bias toward classifying most samples as non-vulnerable. In the case of Go, the models show high accuracy (85% to 93%) but low F1 scores (37% to 55%). While Go has a more balanced dataset compared to PHP, the models still struggle to detect vulnerabilities effectively. This suggests that vulnerability patterns in Go might be more complex or harder to learn with the current model architectures.

The performance of different LMs varies across programming languages, suggesting that no single model architecture consistently outperforms others in all scenarios. Interestingly, the performance rankings of these models do not strictly align with their parameter sizes, showing that factors other than model size contribute to their effectiveness in vulnerability detection.

UnixCoder and CodeBERT, despite having fewer parameters than DeepSeek-Coder, consistently demonstrate strong performance across multiple languages. On the other hand, CodeT5, the smallest model among the four, often lags behind other models in terms of performance across different languages. This indicates that while a larger model size can potentially improve performance, it is not the sole determining factor. DeepSeek-Coder, the largest model, shows mixed performance results across languages. While it achieves the highest accuracy and F1 scores for some languages, such as Java, it does not consistently outperform other models in all cases. This suggests that the relationship between model size and performance is not always linear.

Validation on Other Dataset

To validate the generalizability of the models, it is essential to test their performance on independent datasets. In this study, we evaluated the models trained on the CVEFixes dataset on several independent datasets for different languages.

For Java, the models were tested on the MegaVul dataset. The results show slightly lower performance compared to their performance on the CVEFixes dataset. However, the performance remains consistent across both datasets, with the models maintaining their relative rankings. This consistency in performance provides some validation of the results and suggests that the models are capable of generalizing to new Java vulnerability data to a certain extent.

For PHP, we tested the models on the PHP vulnerability test suite from SARD. The results indicate a drop in performance when compared to the models' performance on the CVEFixes dataset, particularly in terms of Precision and Recall. Additionally, the FPR increased significantly. This can be attributed to the fact that the SARD PHP test cases are synthetic, meaning they were specifically created as examples with well-characterized weaknesses. Each test case targets only one specific flaw. As a result, these test cases have a much simpler structure than most weaknesses found in real-world production code. The models trained on the CVEFixes PHP subset likely learned patterns and characteristics specific to the

real-world vulnerabilities present in that dataset, making it challenging for them to generalize to the synthetic, targeted vulnerabilities in the SARD test cases.

In the case of Python, we evaluated the models on the synth-vuln-fixes dataset. The results show mixed performance, with some models (i.e., CodeT5) maintaining relatively consistent performance across both datasets, while others (i.e., CodeBERT and DeepSeek-Coder) exhibit a significant drop in performance. This indicates that the ability of the models to generalize to new Python vulnerability data may depend on the specific model architecture.

On the other hand, the poor performance of the models in the C/C++ language persists across multiple datasets, including CVEFixes, PrimeVul, DiverseVul, and Gen2Spec. Despite the differences in the datasets' characteristics and the vulnerability types they cover, the models consistently struggle to effectively detect vulnerabilities in C/C++ code. For C/C++, the performance when testing on CVEFixes is better than testing on PrimeVul and other C/C++ datasets, as those datasets have more extensive entries compared to the C/C++ subset in CVEFixes, as shown in Table 4. This suggests that the larger dataset sizes in PrimeVul, DiverseVul, and Gen2Spec may contribute to the lower performance observed when testing on these datasets compared to CVEFixes. However, the overall poor performance across all datasets highlights the inherent difficulty of vulnerability detection in C/C++ and suggests that the challenges are not limited to a specific dataset but are rather a fundamental problem in applying current language models to C/C++ vulnerability detection.

It would be natural to assume that the lower performance of vulnerability detection models for C/C++ compared to other languages may be attributed to the inherent complexity of C/C++ code, as these languages are commonly used in system-level software development. The low-level nature of C/C++, manual memory management, and the potential for more diverse coding styles may introduce challenges for the models to generalize across different codebases. These facts motivate our following second analysis of code complexity versus detection performance.

5.3 RQ2-Results regarding code complexity and detection performance

In this section, we investigate the relationship between code complexity and vulnerability detection performance across the languages we assess. The detection performance is measured using the F1 score of all the LMs for each language. Specifically, we aim to determine whether higher code complexity correlates with lower detection performance, as measured by the model's F1 score. The choice of F1 score is motivated by several factors. First, the F1 score has been widely used in previous vulnerability detection studies, including those focusing on C/C++ [Che+23a; Cha+22b; Din+25]. By using the F1 score, we maintain consistency with prior work and facilitate comparisons across different studies. Second, vulnerability detection datasets often have imbalanced classes (this one applies to our

dataset as well, even though the imbalance is not that high, see table 1), with a higher proportion of non-vulnerable samples compared to vulnerable ones. F1 score is less sensitive to class imbalance than other metrics like accuracy, making it a more suitable choice for evaluating vulnerability detection systems.

Code Complexity Metrics

Evaluating source code complexity can provide insights into maintainability, readability, and potential defect proneness [McC04]. In this work, we use six common metrics to characterize code complexity within each language-specific subset of our dataset:

Token Length Token length is the average number of lexical tokens per snippet. A *token* is a single unit of code as identified by a lexer (i.e., keywords, identifiers, literals, operators). Longer token sequences often indicate more verbose code, although verbosity alone does not necessarily imply higher logical complexity.

Cyclomatic Complexity (CC) Cyclomatic Complexity [McC96] measures the number of linearly independent paths through a program's control-flow graph. Formally,

$$\mathrm{CC} = E - N + 2P, \tag{1}$$

where E is the number of edges, N is the number of nodes, and P is the number of connected components in the graph. Higher cyclomatic complexity suggests more branching and, thus, a greater likelihood of defects and maintenance difficulties.

Halstead Volume (HV) Halstead Volume is one of the Halstead complexity metrics introduced by Maurice Halstead [Hal77]. It quantifies the total size of an implementation. Let η_1 and η_2 be the number of unique operators and operands, respectively, while N_1 and N_2 are the total occurrences of operators and operands. Then the volume is given by:

$$\mathrm{HV} = (N_1 + N_2) \times \log_2(\eta_1 + \eta_2). \tag{2}$$

A high Halstead Volume indicates substantial implementation details and potentially more cognitive load for understanding the code.

Halstead Difficulty (HD) Halstead Difficulty [Hal77] gauges how difficult it is to write or comprehend a given program. It is computed as:

$$\mathrm{HD} = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}, \tag{3}$$

Language	Token Length	Halstead Volume	Halstead Difficulty	Halstead Effort	Cyclomatic Complexity	NLOC
JavaScript	462.05	2204.25	21.83	76395.58	9.46	6.39
PHP	244.44	5.79	0.62	105.03	2.02	1.22
Java	274.79	879.30	10.81	16462.39	3.27	20.50
Python	338.35	862.93	13.07	20805.42	4.36	23.43
Go	752.59	2118.29	23.86	78381.61	7.89	53.57
C/C++	274.79	879.30	10.81	16462.39	3.27	20.50

Table 5: Mean code complexity metrics by programming language.

where η_1 and η_2 are the counts of unique operators and operands, and N_2 is the total number of operands. Intuitively, the more varied or numerous the operands, the higher the difficulty.

Halstead Effort (HE) Halstead Effort [Hal77] builds upon Halstead Volume and Difficulty:

$$\mathrm{HE} = \mathrm{HV} \times \mathrm{HD}. \tag{4}$$

This metric estimates the amount of *mental work* required to implement or understand the code. A high Halstead Effort value suggests that the code is complex and may be more prone to errors.

Number of Lines of Code (NLOC) NLOC simply measures the average number of physical lines per snippet. Although lines of code alone do not capture deeper structural complexity, extremely high or low NLOC values can correlate with maintainability challenges, readability issues, or overly concise "trick" implementations [McC04].

We compute these metrics for each code snippet in our dataset (Section 4), then aggregate them to obtain mean values per language (Table 5). In Section 5.3, we analyze whether these complexity measures correlate with model detection performance (F1 score) across different languages.

Table 5 presents the mean values of these complexity metrics for each programming language we analyze.

Correlation Analysis

We compute the Pearson correlation coefficients between each code complexity metric and the F1 score for each model to assess the strength and direction of their relationships. Table 6 summarizes the correlation coefficients and their corresponding p-values.

The correlation coefficients vary in both magnitude and sign across the different models and complexity metrics. For the CodeT5 model, most of the coeffi-

cients are negative, indicating a slight inverse relationship between code complexity and F1 score, with the exception of NLOC, which shows a positive correlation. However, for the other models (CodeBert, UnixCoder, and DeepSeek-Coder), the coefficients are mostly positive, suggesting a slight positive relationship between code complexity and F1 score. Notably, though, the absolute values of all the coefficients are relatively low, generally below 0.6. Furthermore, the p-values for all correlations are much higher than the typical significance level of 0.05, indicating that none of these correlations are statistically significant. This suggests that the observed relationships between the various code complexity measures and model performance could be due to random chance rather than a true underlying correlation.

Model	Complexity Metric	Correlation Coefficient (r)	P-value
CT5	Token Length	-0.1271	0.8104
	Halstead Volume	-0.0676	0.8988
	Halstead Difficulty	0.0363	0.9456
	Halstead Effort	-0.2002	0.7037
	Cyclomatic Complexity	-0.2347	0.6544
	NLOC	0.3197	0.5368
CB	Token Length	0.0786	0.8823
	Halstead Volume	0.5624	0.2453
	Halstead Difficulty	0.5288	0.2808
	Halstead Effort	0.4339	0.3900
	Cyclomatic Complexity	0.5177	0.2928
	NLOC	-0.0681	0.8980
UC	Token Length	0.3619	0.4808
	Halstead Volume	0.4558	0.3637
	Halstead Difficulty	0.4371	0.3861
	Halstead Effort	0.4853	0.3292
	Cyclomatic Complexity	0.5336	0.2756
	NLOC	0.0555	0.9169
DS-C	Token Length	-0.0431	0.9354
	Halstead Volume	0.3435	0.5050
	Halstead Difficulty	0.3240	0.5310
	Halstead Effort	0.2499	0.6330
	Cyclomatic Complexity	0.3469	0.5006
	NLOC	-0.1693	0.7485

Table 6: Correlation coefficients between complexity metrics and F1 score.

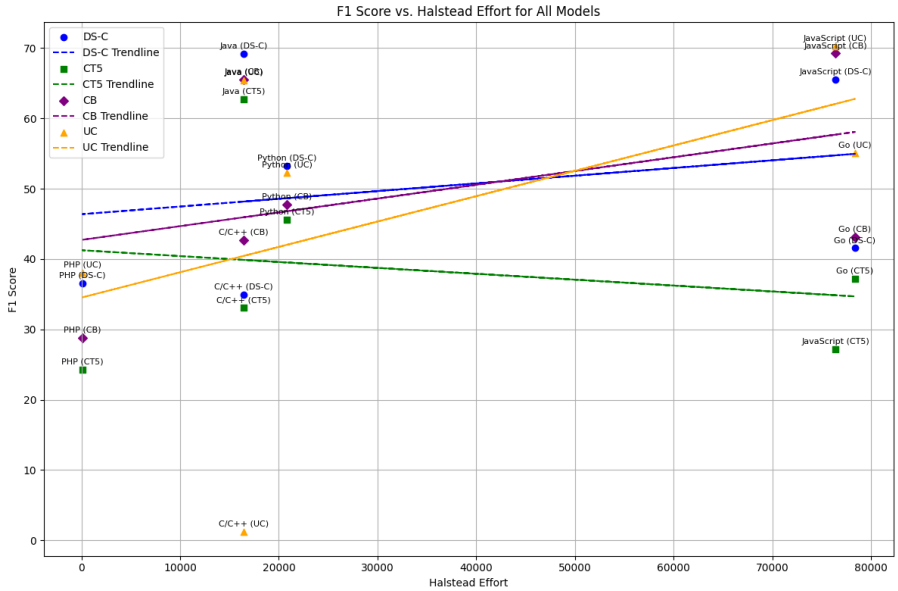

Figure 2: Mean Halstead effort vs. F1 score for each programming language.

Visualization of Results

To further explore the relationships, we plot the mean F1 scores against key complexity metrics, specifically Halstead Effort and Cyclomatic Complexity. In Figure 2, which shows F1 scores versus Halstead Effort, three of the trendlines (DS-C, CB, and UC) have a positive slope, while the CT5 trendline slopes downward. This mixture of upward and downward trends, along with the dispersed data points, suggests that no uniform relationship emerges. Similarly, Figure 3 plots F1 scores against Cyclomatic Complexity, and here again, DS-C, CB, and UC trendlines slope upward, whereas CT5 slopes downward. Overall, these mixed directions and the wide spread of points indicate that any correlation between these complexity metrics and F1 scores is weak and inconsistent across the different models.

6 Limitation

Our study relies on the CVEfixes dataset, which is comprehensive and diverse, covering a significant portion (∼80%) of the available vulnerability fixes from the NVD database. While the CVEfixes dataset as a whole is extensive, the dataset sizes for individual languages like Go, Java, and Python are relatively small compared to the C/C++ datasets used in previous studies, such as DiverseVul and PrimeVul. These smaller dataset sizes are a result of the inherent distribution of vulnerabili-

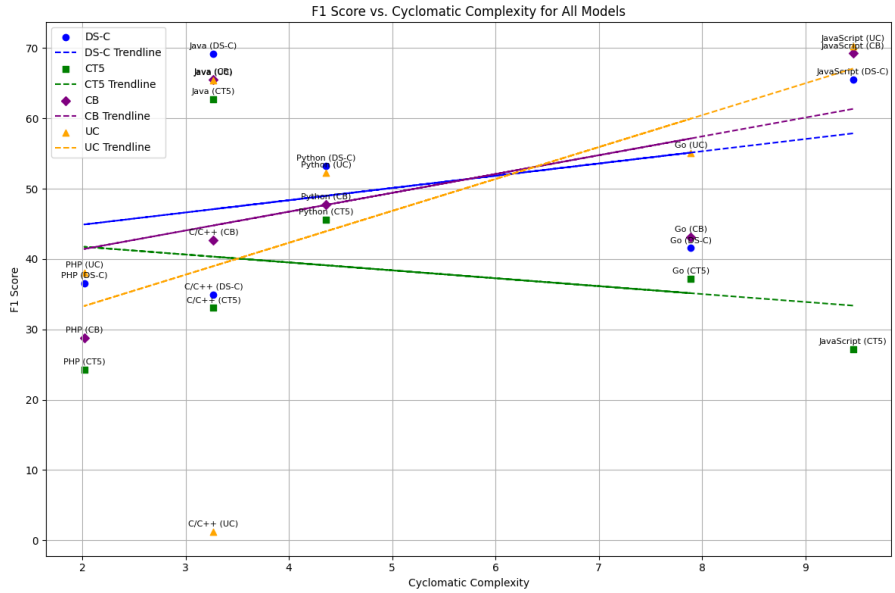

Figure 3: Mean cyclomatic complexity vs. F1 score for each programming language.

ties across languages in the NVD database (real-world representation) and not a limitation of the CVEfixes dataset itself. The CVEFixes paper acknowledges that at the time of writing, ∼80% of fixes in the NVD database refer to code on one of the three major forges: GitHub, GitLab, and Bitbucket, with 98% of those being on GitHub. The remaining 20% of fixes point to other forges (i.e., SourceForge or the defunct Gitorious) or to project-specific servers hosting other versioning systems (i.e., Mercurial, Subversion, or CVS). One could expand the dataset by manually locating and collecting data from projects that are not listed on GitHub, GitLab, or Bitbucket. This process would require manual collection and verification efforts.

However, even without the code hosted on non-GitHub, GitLab, and Bitbucket, we believe our investigation's results would not change significantly, as they only constitute the remaining ∼20% of the total data.

7 Conclusions

In this study, we investigated the effectiveness of multiple language models (LMs) for vulnerability detection across several popular programming languages, including JavaScript, Java, Python, PHP, Go, and C/C++. Our results demonstrate that the performance of LMs varies depending on the language, with models gen-

erally performing better on non-C/C++ languages compared to previous work on C/C++. We found that vulnerability detection models for languages such as JavaScript and Java achieve higher F1 scores compared to the results reported in prior studies on C/C++. This suggests that LMs can be more effective and practical for vulnerability detection in these languages. However, performance varies between different languages, with PHP and Go showing high accuracy but lower F1 scores.

We also explored the relationship between code complexity and vulnerability detection performance across the studied languages. Our analysis revealed weak and statistically insignificant correlations between various complexity metrics and the models' F1 scores. This suggests that code complexity, as measured by these metrics, may not be a strong predictor of LMs' vulnerability detection performance.

We evaluated the generalizability of our findings on independent datasets for several languages. For Java, results on the MegaVul dataset closely mirrored those obtained from CVEFixes, suggesting consistent performance across different data sources. In contrast, testing C/C++ models on the PrimeVul dataset reaffirmed the difficulty of achieving robust vulnerability detection performance in these languages. For Python, performance on the synth-vuln-fixes dataset showed mixed results, indicating that some models struggle to generalize effectively. Similarly, testing the PHP models on the SARD dataset revealed challenges in detecting synthetic, well-characterized vulnerabilities as opposed to the more diverse real-world samples found in CVEFixes.

Our study has limitations that should be acknowledged. While the CVEfixes dataset is comprehensive, covering 80% of the available vulnerability fixes from the NVD database, the dataset sizes for individual languages like Go, Java, and Python are relatively small compared to C/C++. Expanding the dataset to include the remaining 20% of fixes from other platforms would require manual collection and verification efforts. However, we believe that the absence of this data does not significantly impact our findings.

Acknowledgements

This work is partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation and by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research. The computations and data handling were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

References

[Alo+19] U. Alon et al. “code2vec: learning distributed representations of code.” In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019).

[Ati+24] S. A. Atiiq et al. *From Generalist to Specialist: Exploring CWE-Specific Vulnerability Detection*. 2024. arXiv: 2408.02329 [cs.CR].

[Bal99b] T. Ball. “The concept of dynamic analysis.” In: *SIGSOFT Softw. Eng. Notes* 24.6 (Oct. 1999), pp. 216–234.

[BB22] L. Braz and A. Bacchelli. “Software security during modern code review: the developer’s perspective.” In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 810–821.

[BNM21a] G. Bhandari, A. Naseer, and L. Moonen. “CVEfixes: automated collection of vulnerabilities and their fixes from open-source software.” In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 30–39.

[Cha+22b] S. Chakraborty et al. “Deep Learning Based Vulnerability Detection: Are We There Yet?” In: *IEEE Transactions on Software Engineering* 48.9 (2022), pp. 3280–3296.

[Che+23a] Y. Chen et al. “DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection.” In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 654–668.

[CM04] B. Chess and G. McGraw. “Static analysis for security.” In: *IEEE Security & Privacy* 2.6 (2004), pp. 76–79.

[Dai+24] D. Dai et al. “DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models.” In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1280–1297.

[Dev+19a] J. Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186.

[Din+25] Y. Ding et al. "Vulnerability Detection with Code Language Models: How Far Are We?" In: *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 469–481.

[Fen+20a] Z. Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by T. Cohn, Y. He, and Y. Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[Guo+21] D. Guo et al. "GraphCode{BERT}: Pre-training Code Representations with Data Flow." In: *International Conference on Learning Representations*. 2021.

[Guo+22] D. Guo et al. "UniXcoder: Unified Cross-Modal Pre-training for Code Representation." In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by S. Muresan, P. Nakov, and A. Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225.

[Guo+24b] D. Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE].

[Hal77] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. USA: Elsevier Science Inc., 1977.

[LE20] J. Lumbroso and J. Evans. "Making Manual Code Review Scale." In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: Association for Computing Machinery, 2020, p. 1390.

[Li+18a] Z. Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection." In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018.

[Li+24] H. Li et al. “Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach.” In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024).

[Lia+25] C. Liang et al. “Survey of source code vulnerability analysis based on deep learning.” In: *Computers & Security* 148 (2025), p. 104098.

[Lin+20] G. Lin et al. “Software Vulnerability Detection Using Deep Neural Networks: A Survey.” In: *Proceedings of the IEEE* 108.10 (2020), pp. 1825–1848.

[Lin+24] R. Lin et al. “Vulnerabilities and Security Patches Detection in OSS: A Survey.” In: *ACM Comput. Surv.* (Sept. 2024). Just Accepted.

[Liu+20] Y. Liu et al. *Ro{BERT}a: A Robustly Optimized {BERT} Pretraining Approach*. 2020.

[Lu+21a] S. Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021. arXiv: 2102.04664 [cs.SE].

[McC04] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2nd. Microsoft Press, 2004.

[McC96] T. McCabe. “Cyclomatic complexity and the year 2000.” In: *IEEE Software* 13.3 (1996), pp. 115–117.

[McG04] G. McGraw. “Software security.” In: *IEEE Security & Privacy* 2.2 (2004), pp. 80–83.

[Nam+24] D. Nam et al. “Using an LLM to Help With Code Understanding.” In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Lisbon, Portugal: Association for Computing Machinery, 2024.

[Ni+24] C. Ni et al. “MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations.” In: *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 2024, pp. 738–742.

[Nik+21b] G. Nikitopoulos et al. “CrossVul: a cross-language vulnerability dataset with commit data.” In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1565–1569.

[Rad+19] A. Radford et al. “Language models are unsupervised multitask learners.” In: *OpenAI blog* 1.8 (2019), p. 9.

[Ros+24] M. Rosenblatt et al. “Data leakage inflates prediction performance in connectome-based machine learning models.” In: *Nature Communications* 15.1 (2024), p. 1829.

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask).” In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 317–331.

[Sad+18b] C. Sadowski et al. “Modern code review: a case study at google.” In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 181–190.

[Sca+09] F. Scarselli et al. “The Graph Neural Network Model.” In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80.

[SF16] B. Stivalet and E. Fong. “Large Scale Generation of Complex and Faulty PHP Test Cases.” In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, pp. 409–415.

[Shi+24] K. Shi et al. *Natural Language Outlines for Code: Literate Programming in the LLM Era*. 2024. arXiv: 2408.04820 [cs.SE].

[Svy+20] A. Svyatkovskiy et al. “Intellicode compose: Code generation using transformer.” In: *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 1433–1443.

[Tea24] G. Team. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL].

[Tem+24] A. Templeton et al. *Scaling monosemanticity: extracting interpretable features from Claude 3 Sonnet, Transformer Circuits Thread*. 2024.

[Tho21] P. Thomson. “Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity.” In: *Queue* 19.4 (Sept. 2021), pp. 29–41.

[Tym17] Y. Tymchuk. “The False False Positives of Static Analysis.” In: (2017).

[Ull+24] S. Ullah et al. “LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks.” In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 862–880.

[Vas+17] A. Vaswani et al. “Attention is All you Need.” In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. New York, NY, USA: Curran Associates, Inc., 2017.

[Wan+21c] Y. Wang et al. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.” In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens et al. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708.

[Yan+22] Y. Yang et al. “A Survey on Deep Learning for Software Engineering.” In: *ACM Comput. Surv.* 54.10s (Sept. 2022).

[Zho+19a] Y. Zhou et al. “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.” In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

Popular Science Summary

Popular Science Summary

We live in a world where everything, from the phone in your pocket to the traffic lights on your commute, relies on having a stable, always-on connection. When these systems go offline, even briefly, daily life can be disrupted. During major crises like the COVID-19 pandemic, we became even more aware of how crucial it is for digital systems to keep working, whether for remote work, online learning, or maintaining critical services such as hospital networks and emergency call centers.

This dissertation focuses on how to keep modern digital systems, particularly the so-called Internet of Things (IoT) and the latest 5G mobile networks, up and running despite failures, attacks, or software bugs.

Safeguarding the Internet of Things

"Internet of Things" is a broad term for devices that connect to the internet and sense or control the world around us. These devices range from household items like "smart" doorbells and security cameras to industrial equipment that monitors factories or power grids. Because IoT devices are often tiny, low-power gadgets, they are very sensitive to disruptions. A malicious attacker can overwhelm them by sending too much data, a type of assault called a Denial of Service attack. The first part of this dissertation explores new ways to stop these attacks before they turn off vital functions. For example:

- CLI-DOS and X-Pro are security techniques that share information between devices or rely on special network checkpoints to block harmful internet traffic. This stops waves of malicious data from ever reaching the more vulnerable IoT devices.
- A system called Lazarus was improved to help IoT devices reset themselves safely after severe attacks. Even if an attacker almost completely takes over a device, our improved version of Lazarus helps it recover, ensuring it can continue working.

Strengthening 5G Networks

The fifth generation (5G) of cellular networks promises faster speeds, lower delays, and better support for connected devices. However, this added complexity

can create new risks. One new part of 5G, called the Network Data Analytics Function (NWDAF), uses artificial intelligence to predict how people and devices will move around the mobile network. This knowledge helps network operators offer smoother handovers and better coverage. Unfortunately, attackers could trick the system with fake or misleading data, making it much harder to plan network resources effectively. Research in this dissertation finds:

- Even a small group of fake devices can drastically reduce the accuracy of NWDAF's AI predictions.
- With clever retraining of the AI models or smarter ways to pick which model to use, these attacks can be made less harmful.

Finding Software Weak Spots with Artificial Intelligence

Every modern digital system, whether it's an IoT sensor or a 5G base station, runs on software. Sometimes, that software contains hidden flaws that hackers can exploit to crash systems or steal data. Tools known as code language models (think of them as "spellcheckers" for coding) can automatically scan programs to detect these vulnerabilities. By looking at gigantic samples of real code, these tools learn to spot dangerous programming patterns. Studies in this dissertation reveal:

- While these AI models often perform better than older methods, they still struggle to detect all vulnerabilities in complex real-world code.
- Performance varies across programming languages; some are easier for AI to check than others.

Why It All Matters

Our world depends on computers and networks for everything from social media to life-saving hospital equipment. If hackers manage to shut down these systems, the consequences can be huge, both socially and economically. The research here introduces new ways to block attacks, recover from failures, and spot software bugs before they become serious threats. By making IoT devices, 5G networks, and the software they rely on more robust, we can help ensure our increasingly connected society continues to function securely, even under pressure.