

UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

IMPROVING VULNERABILITY TESTING AND DETECTION IN THE EMBEDDED CONTEXT

Davide Rusconi

ORCID 0009-0009-3648-7416

PhD program in Computer Science – Cycle XXXVIII

Tesi redatta con il contributo finanziario dell'Unione europea – Next Generation EU
e Security Pattern S.R.L CUP n. G43C22002330004

Supervisor: Prof. **Danilo Bruschi**

Co-Supervisor: Prof. **Andrea Lanzi**

Coordinator: Prof. Roberto Sassi

Academic year
2024-2025

Pour ceux qui suivront

Acknowledgements

The end of my PhD journey marks the end of an era in my life. Looking back at these three years, I fondly remember all the people I've met and without whom I would not have been able to overcome the challenges I faced. This work came to be thanks to all the people who supported me throughout this process, both by guiding my research directly and by supporting me with their friendship. Before starting, I want to give these people the spotlight they deserve; without all of you, this would not have been possible.

I want to start by expressing my thanks to my parents, who have always been supportive of my choices and without whom I would not be who I am today. I cannot put into words how grateful I am to Alessia, who always stood by my side, helped me at my lowest, and encouraged me at my highest. I want to thank Luigi, Michele, Filippo, Arianna, Luca, and Stefano: you've been by my side well before this PhD, and I hope we will stay together well beyond it. I also want to thank Manuel, who was always there when everyone else was far away; I hope I can repay at least a fraction of the help you gave me. I also want to thank all the friends I made along the way, who made these years as memorable as they have been.

Finally, I want to thank Andrea Lanzi and Danilo Bruschi for setting this PhD in motion and for their guidance throughout. A big thank you to Davide Balzarotti, without whom I would never have worked on fuzzing, a topic I came to love, and I would have missed out on the valuable experience of my visit to EURECOM. I am grateful to all my colleagues and co-authors who helped polish and refine my ideas. And a final thanks to Security Pattern, that chose to invest in me by co-funding my PhD and enabled me to bridge academia and industry.

Abstract

Embedded systems are becoming a pervasive technology in today’s computational and societal landscape, with over 17 billion active devices in 2024 and projections exceeding 29 billion by 2030. Their applications span a wide range of domains, including both safety- and security-critical contexts. With a number of 5000 IoT-directed attacks per month and a cost estimate of 500,000\$ per attack, enhancing both pre-deployment vulnerability attestation through automated testing and runtime vulnerability detection mechanisms represents not only a practical necessity but also a significant research challenge, given the limited reusability of existing solutions in embedded contexts. This thesis addresses both of these challenges by proposing novel, tailored solutions. First, it introduces a low-overhead firmware instrumentation framework designed for the runtime detection of spatial memory errors in embedded software, along with mechanisms for root-cause localization of such violations. Second, it presents two contributions to the field of automated vulnerability detection via fuzz testing. The first addresses the scalability challenges of in-place embedded fuzzing by proposing a novel architecture that enables pseudo-parallelization of the fuzzing process to maximize testing throughput. The second contribution tackles the current limitations of fuzzing bugs as a whole by systematically modeling the features that make defects difficult to detect. Based on this model, a fuzzing-based measurement framework is introduced and applied to both state-of-the-art benchmarks and real-world vulnerabilities, enabling the analysis of potential biases toward specific bug classes.

Contents

Acknowledgements	iii
1 Introduction	1
1.1 Open Problems	2
1.2 Contributions	4
2 Background and Related Work	7
2.1 Vulnerability Detection	7
2.2 Vulnerability Testing	12
2.2.1 Fuzz Testing	12
2.2.2 Bug Complexity	16
3 Efficient runtime detection of spatial memory errors in embedded systems	18
3.1 Problem Statement	18
3.2 Threat Model	20
3.3 EMBEDWATCH Overview	21
3.3.1 Running Example	21
3.3.2 System Architecture	22
3.4 EMBEDWATCH Protocol	24
3.4.1 The Protocol in a Glance	24
3.4.2 Graph Data Modeling	25
3.4.3 Firmware Instrumentation	28
3.4.4 Loop Optimization	29
3.4.5 Trace Validation	29
3.5 Evaluation	32
3.5.1 RQ1: Bug Analysis & False Positives	33
3.5.2 RQ2: System Usability & Performance	34
3.5.3 RQ3: Security Analysis	38
3.6 Discussion	39
3.7 Conclusion	41

4	Optimizing Embedded Device Security with Scalable In-Place Fuzzing	42
4.1	Problem Statement	42
4.2	E-FuzzEdge Design	44
4.2.1	Architectural Model	45
4.2.2	Communication Protocol	46
4.3	Implementation	47
4.4	Evaluation	47
4.4.1	Dataset	48
4.4.2	Results	48
4.5	Discussion	50
4.6	Conclusion	51
5	Bug Complexity and their presence in fuzzing benchmarks	53
5.1	Problem Statement	53
5.2	Taxonomy of bug complexity	56
5.2.1	Defining Bugs	56
5.2.2	Defining Bug Complexity	57
5.2.3	A Formal Framework for Bug Triggering	57
5.2.4	Illustrative Example	59
5.2.5	Bug Complexity Indicators	59
5.3	Methodology	60
5.3.1	Program Instrumentation	61
5.3.2	Targeted Fuzzing Campaigns	61
5.4	Implementation	63
5.4.1	Bug Reproduction and Deduplication	64
5.5	Experiments	65
5.5.1	MAGMA	67
5.5.2	OSS-FUZZ	68
5.5.3	Edge dependency	69
5.6	Discussion	71
5.7	Limitations & Threats to Validity	73
5.8	Conclusion	75
6	Conclusion	79

Chapter 1

Introduction

Embedded systems are becoming increasingly pervasive in modern society, with forecasts predicting the deployment of tens of billions of such devices by 2030 [97]. These systems are found in an extremely broad range of contexts, from consumer electronics to safety-critical infrastructure like industrial control systems, transportation equipment, and medical devices.

This ubiquity makes embedded systems both highly valuable and highly vulnerable. Their proper functioning is essential not only for user convenience but also for the safety and reliability of essential services. At the same time, their network connectivity exposes them to a broad class of remote attacks. Their long lifespans, lack of maintenance, and frequent absence of strong isolation mechanisms make them particularly attractive to attackers.

The potential impact of vulnerabilities in embedded systems is extremely severe, with data showing a cost of approximately 500,000\$ per attack and an average of 5000 attacks per month targeting this class of systems [80, 33]. One notable example is Stuxnet, a worm developed jointly by the United States and Israel. The malware was engineered to compromise programmable logic controllers (PLCs) in Iranian nuclear facilities, modifying centrifuge behavior while concealing its actions from monitoring systems. The resulting damage slowed Iran's nuclear enrichment program. Another example is the Mirai botnet, which compromised consumer-grade Internet-of-Things (IoT) devices with weak credentials and repurposed them to launch massive distributed denial-of-service (DDoS) attacks. These examples highlight the dual role of embedded systems in modern cyber threats: they are targets of both sophisticated nation-wide operations and large-scale opportunistic exploitation of motivated individuals.

Despite the increasing relevance of embedded system security, protecting these devices remains a challenge. Unlike general-purpose systems, embedded platforms are characterized by strict resource constraints, heterogeneity in both hardware and software, and a frequent absence of modern oper-

ating system features such as memory isolation or runtime introspection. These constraints make it difficult to directly apply many defensive techniques developed in the context of desktop or server environments and their heterogeneity complicates the development of one-fits-all security solutions.

The process of securing a computer system, be it embedded or general purpose, can follow complementary approaches. The first is pre-deployment testing, which aims to identify and fix software vulnerabilities before the system is released into the field. The second is runtime protection, which involves detecting and reporting vulnerabilities that manifest after deployment, including those that were missed during testing. Both approaches play a vital role: pre-deployment testing contributes to safer releases, while runtime protection enables fault detection, diagnostics, and postmortem analysis after real-world failures.

Fuzzing has become the de facto standard for automated vulnerability discovery during pre-deployment testing. By generating and executing large volumes of malformed or unexpected inputs, fuzzers aim to uncover crashes or logic errors that signal software flaws. However, fuzzing embedded systems introduces additional challenges. Many devices cannot easily be instrumented, lack operating system support, and run on custom hardware with limited interfaces and resources.

Runtime detection for embedded systems faces a different set of obstacles. Techniques must operate with minimal performance overhead and must not rely on features that are often unavailable, such as virtual memory or exception handling. In addition, embedded platforms vary widely in their hardware architectures and software stacks, making it difficult to develop general-purpose detection mechanisms. As a result, existing runtime protections either require intrusive modifications, incur unacceptable slowdowns, or cannot be deployed on bare-metal systems.

This thesis addresses these limitations by proposing new approaches for both runtime vulnerability detection and dynamic testing in embedded systems. Our contributions span three complementary directions. First, we introduce a runtime detection system tailored to embedded firmware, capable of reporting precise crash information with negligible overhead. Second, we present a high-throughput, in-place fuzzing architecture designed to support effective and scalable fuzzing campaigns on real hardware platforms. Finally, we investigate the notion of bug complexity in the context of fuzzing and develop a methodology for evaluating it across multiple real-world benchmarks.

1.1 Open Problems

Despite the growing body of research in embedded security, both runtime detection and fuzzing remain fundamentally challenging in this domain.

For runtime detection, the first and most immediate issue is performance overhead. Embedded systems often operate under strict timing constraints and are built with low-power, low-frequency processors. In this context solutions that impose even moderate CPU, memory, or I/O overhead may render the system unusable or violate real-time constraints in embedded contexts. Additionally, these systems often lack critical software abstractions like virtual memory, memory protection units, or comprehensive exception handling, which are typically leveraged by defensive tools. As a result, runtime detection mechanisms must be carefully designed to work in resource-constrained, low-feature environments without sacrificing effectiveness.

Another central difficulty lies in the extreme heterogeneity of embedded platforms. Devices vary widely in terms of processor architectures, available instruction sets, peripheral configurations, and software stacks. Some run full operating systems like embedded Linux, while others operate entirely without an OS. Designing detection systems that can generalize across such diversity is particularly difficult, especially when trying to minimize platform-specific assumptions or dependencies.

Fuzz testing also presents several open problems, some of which span both general-purpose and embedded-specific dimensions. One unresolved issue in the broader fuzzing landscape is how to meaningfully evaluate a fuzzer’s effectiveness. While metrics such as code coverage or number of crashes are commonly used, they fail to capture the qualitative nature of the bugs found. Recent work has shown that different fuzzers often discover different types of vulnerabilities [46], raising the concern of possible feedback loops in the current fuzzing evaluation frameworks. This complicates both the development and comparison of fuzzers, and can hinder fuzzing research across all targets.

In the embedded domain, these challenges are compounded by practical constraints. Fuzzing can be performed either through emulation-based methods or via in-place testing on the actual hardware. Emulation enables scalability by running multiple instances of the firmware in parallel on host machines, potentially accelerating input generation and feedback collection. However, emulating hardware peripherals and interactions is notoriously difficult and error-prone. Many embedded devices offload complex functionality to specialized hardware accelerators or custom peripherals, which are difficult to model accurately. Inaccurate models can lead to missed bugs (false negatives) or spurious crashes (false positives).

In-place fuzzing, by contrast, ensures realism by running the firmware natively on the target device. However, it suffers from significant scalability limitations. Executing inputs directly on hardware is slow, communication channels are often narrow or unstable, and increasing throughput requires physical duplication of the test setup. These limitations make high-throughput, feedback-driven fuzzing difficult to achieve in practice.

Together, these problems motivate the need for efficient, accurate, and

scalable techniques for both fuzzing and runtime vulnerability detection in embedded systems.

1.2 Contributions

This thesis addresses the above challenges through three core contributions that aim to answer the following research questions:

- **RQ1** *How can we accurately assess and report memory safety violations on embedded devices at runtime, despite tight resource constraints and limited observability?*
- **RQ2** *How can the execution throughput of hardware-based fuzzing be improved without relying on peripheral emulation or other unrealistic modelling assumptions?*
- **RQ3** *Which intrinsic properties of real-world vulnerabilities limit the ability of coverage-guided fuzzers to reach or trigger them in practice?*

The work starts with a background section in Chapter 2 followed by one chapter for each contribution indicated in this section.

The first contribution, presented in Chapter 3, introduces a runtime system for detecting spatial memory violations in embedded firmware. The system combines a lightweight memory instrumentation strategy based on fat pointers with remote attestation to enable the detection and classification of out-of-bounds memory accesses across stack, heap, and global regions. In contrast to prior work that focused solely on control flow attestation our system is capable of providing guarantees comparable to general-purpose solutions targeting the same problem while operating with minimal assumptions and incurring in negligible performance overhead. By performing fine-grained fault attribution, it also enables actionable postmortem vulnerability analysis for real-world firmware.

In Chapter 4 we present E-FuzzEdge, a novel fuzzing architecture for in-place testing on real embedded devices. Unlike emulation-based approaches, which often struggle to model device-specific peripherals, E-FuzzEdge performs fuzzing directly on the target hardware. To address scalability and performance bottlenecks, the architecture is optimized for high-throughput execution and integrates lightweight communication and feedback mechanisms. Our evaluation demonstrates that E-FuzzEdge significantly improves fuzzing throughput and reliability in resource-constrained settings. The architecture is designed to be easily composable with other fuzzing frameworks and can serve as a drop-in backend for state-of-the-art on-device fuzzing workflows.

Finally, Chapter 5 turns the attention to the problem of evaluating fuzzers and understanding why they succeed or fail. We propose a novel

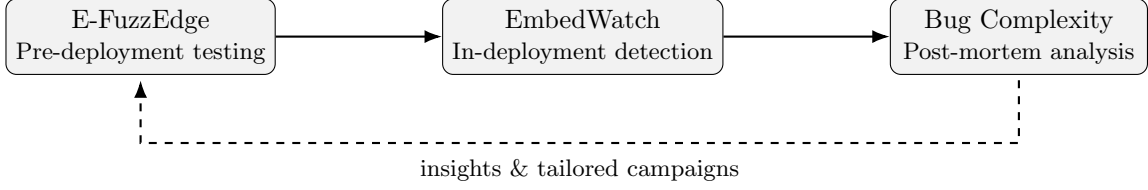


Figure 1.1: Overview of the proposed framework integrating testing, monitoring, and analysis across the embedded firmware life cycle.

taxonomy of bug complexity that decomposes vulnerabilities into three orthogonal dimensions: reachability, path sensitivity, and state dependence. Through a series of controlled fuzzing campaigns across MAGMA and OSS-Fuzz bugs, we empirically validate this framework and demonstrate that path complexity is a dominant factor in fuzzing success, while deep state dependencies are relatively rare. These findings provide insight into the limitations of current fuzzing strategies and inform the design of more representative benchmarks and evaluation methodologies.

Together, these contributions define an iterative methodology aimed at strengthening the security of embedded devices throughout the entire firmware lifecycle: from pre-deployment testing, to in-deployment vulnerability detection, and post-mortem improvement. Their interaction follows a logical sequence:

1. **E-FuzzEdge** is employed during the pre-deployment stage as a security testing tool on release candidates. Its capability to execute tests directly on the device increases testing fidelity and significantly reduces false positives, improving the overall robustness of the system before market release.
2. **EmbedWatch** becomes relevant once the product is deployed. This system monitors and protects the device during normal operation, providing detailed insight into software faults and security incidents. By enhancing the device’s ability to detect faults and attacks in real time, EmbedWatch contributes to security in two complementary ways: it can detect and mitigate zero-day vulnerabilities as soon as they manifest, and it shortens the time-to-fix for discovered bugs, thereby reducing the vulnerable exposure window of affected devices.
3. Finally, the **Bug Complexity Framework** supports post-mortem analysis, guiding future pre-deployment testing. Once an incident has been identified by EmbedWatch, the framework can evaluate the underlying complexity metrics of the discovered vulnerability to understand why it evaded earlier testing. These insights can then be used to refine the testing strategy or adjust the configuration of E-FuzzEdge,

for example, by launching new fuzzing campaigns with feedback tailored to the missed bug class, thereby improving future detection capability and continuously increasing firmware resilience over successive release cycles.

In summary, although each contribution improves a different aspect of embedded system security, together they form a cohesive process that enhances the overall resilience of these devices beyond the sum of their individual parts.

Chapter 2

Background and Related Work

In this chapter, we present background material on vulnerability detection and fuzzing, both in the general-purpose and embedded context. We also discuss IoT attacks and hardware-specific countermeasures. Alongside this introduction, we review the most relevant works in each area to provide the necessary context for the contributions of this thesis.

2.1 Vulnerability Detection

Before examining state-of-the-art detection solutions, it is important to identify the types of threats that embedded firmware faces in real-world deployments, as well as the tools that can be leveraged to mitigate them.

IoT Attacks The current threat model for embedded systems can be decomposed into two macro classes: hardware-based attacks and software-based attacks. The first category includes attack vectors that require access to the hardware. These types of attacks can either examine hardware features, such as the power consumption or the emission of EM waves, to infer data computed by the system or directly inject anomalies, e.g. voltage spikes, at runtime in order to bring the system into a faulty state and exploit it. The second class comprises software-based attacks that aim at compromising the embedded firmware via malevolent inputs. Recently, advanced attacks, such as return-oriented programming (ROP) and data-only attacks, have also emerged as a threat to embedded devices, particularly those using the ARM architecture [66]. Given the poor security of these devices, it is advisable for IoT backends to assume that IoT devices in the field may be compromised and should not be fully trusted. Even though tools [93, 15] that identify memory attacks can be helpful, they often lack the ability to

identify the root cause of vulnerabilities. This is a crucial limitation, as understanding the underlying cause of a vulnerability is essential to effectively address the issue.

ARM Trustzone In this work ARM TrustZone technology will be utilized to establish a Trusted Computing Base (TCB) [60]. TrustZone is a hardware feature that is available on both Cortex-A processors, used in mobile and high-end IoT devices, and Cortex-M processors, used in low-cost embedded systems. ARM TrustZone is a security extension to the ARM processor architecture that enables the creation of a secure environment to execute sensitive operations. It allows for the separation of a device’s normal and secure worlds, where the normal world contains untrusted applications, and the secure world contains trusted applications and a small operating system.

The secure world is an isolated environment with specialized caches, banked registers, and private memory, which ensures that trusted applications and data are protected. TrustZone also enables the provision of per-device private keys and certificates, which facilitates secure communication and authentication. The normal world can access the secure world through a set of secure monitor calls (SMCs), which allow the normal world to request sensitive services from the secure world, such as signing or securely storing data. TrustZone is widely used in mobile devices, secure elements, IoT devices, and embedded systems to provide a secure environment for sensitive operations. However, while TrustZone provides strong isolation, it does not prevent vulnerabilities in normal-world firmware, which remain the primary target of this thesis.

Having outlined the threat landscape of embedded systems and the hardware-based countermeasures available (e.g., ARM TrustZone), we now turn to prior research that addresses software-level protection and analysis techniques. In particular, since the contribution presented in Chapter 3 builds on a remotely-attested fat-pointer system with the capability of pinpointing the location of identified bugs, we first review the state-of-the-art of fat-pointer-based memory-safety mechanisms before examining runtime remote attestation protocols and root-cause analysis systems. Reviewing these works provides the necessary context to understand the limitations of current approaches and to motivate our contribution to runtime vulnerability detection.

Fat Pointer System Fat pointer systems, like Safe-C [21], CCured [78], and Cyclone [61], combine pointer values with associated bounds metadata into a single structure, differing in how they store this metadata. An alternate approach to managing bounds metadata is through shadow memory, used in memory safety systems like SoftBound [77], PAriCheck [104], Baggy Bounds Checking [17] and Intel MPX. This method links objects or

pointers in the main memory with metadata stored in a separate shadow memory, creating a relationship between them. Recent developments have introduced “low-fat pointers”, particularly effective in 64-bit systems with extensive pointer bit-width [44]. This innovative concept embeds bounds metadata directly within the pointer’s representation, contrasting with fat pointer or shadow space methods that store metadata in a separate location. Implementing fat-pointer techniques in embedded systems encounters several challenges. The main issue is that the instrumentation and runtime checking of pointers required by these systems can significantly slow down execution, making them inefficient for embedded systems, which often have limited resources. To address these challenges, optimization must be devised to minimize performance overhead without significantly impacting detection capabilities. One such optimization is to offload detection to a different system through Remote Attestation, a technique discussed in the following section.

Temporal Memory Safety Temporal memory safety remains one of the most elusive and dangerous classes of vulnerabilities. Unlike spatial violations, which can be checked locally by comparing a memory operation against known object bounds, temporal violations are not a local property. Whether an access is valid depends not only on the instruction performing it, but also on the entire sequence of allocation and deallocation events that preceded that point in the execution. This inherent dependence on object lifetime and reuse semantics makes temporal memory errors substantially harder to detect than spatial ones, especially in systems where precise lifetime metadata is unavailable or cannot be maintained efficiently.

In general-purpose systems, several strategies have been proposed to enforce temporal memory safety. One family of approaches performs explicit tracking of object lifetimes at runtime, often via metadata or alias structures that let the system detect accesses to freed objects, as in RTT-UAF [43] and DangZero [52]. Another line of work infers temporal correctness by analysing how references are used over time: FREEWILL [58] diagnoses use-after-free bugs by detecting reference miscounting at the binary level. A third group modifies allocation behaviour itself, for example by never or rarely reusing freed virtual addresses, or by overprovisioning and randomising the heap layout to reduce the chance that stale pointers collide with newly allocated objects, as implemented in FFmalloc [99] and DieHard [23]. Complementary to these runtime defences, static analyses such as UAFX [108] identify use-after-free vulnerabilities ahead of time by reasoning about cross-entry aliasing in large codebases.

However, these techniques come at a high cost in terms of runtime overhead, memory usage, and reliance on system-level features (e.g., virtual memory, page tables), making them fundamentally unsuitable for low-power,

resource-constrained microcontrollers.

To illustrate this mismatch, DangZero detects UAFs by accessing page table metadata to trace memory usage efficiently, a technique impossible to implement on bare-metal MCUs without an MMU. FFmalloc, on the other hand, introduces a custom allocator that enforces a “one-time allocation” policy to ensure freed memory is never reused, thereby preventing UAFs entirely. While effective, this level of allocator control is impractical in embedded systems where developers often rely on lightweight or custom heap implementations optimized for real-time constraints.

Runtime Remote Attestation of Attacks C-FLAT [15] is the first remote attestation protocol designed to verify the runtime correctness of embedded devices. It focuses on measuring control flow integrity and targets control flow attacks. This protocol has inspired future runtime remote attestation protocols, such as LO-FAT [39], which is a hardware implementation of C-FLAT that improves performance. ATRIUM [106] extends the protection offered by both C-FLAT and LO-FAT against physical attacks, that is, adversaries who physically tamper with the embedded device. After C-FLAT, new protocols were developed to address data-only attacks, such as LiteHax [38]. LiteHax measures firmware data-flow and leverages symbolic execution for verification, but it has a prohibitive overhead that makes it unsuitable for production environments. DIAT [16] proposes a coarse-grained data flow mechanism to verify the correct data transmission among the firmware modules and pairs it with a lightweight control flow attestation.

Among the notable works in the field of remote runtime attestation, OAT [93] and a recent refinement BLAST [103] emerge as the latest advancement, encompassing control flow verification and data-only verification for ARM-based embedded devices. However, these systems primarily focus on attack detection, identifying anomalies consistent with their respective attacker models. Regrettably, all these systems fail to provide crucial information on the location of vulnerabilities, leaving analysts and developers without the necessary insight to develop effective patches.

Another system that recently surfaced is IPEA [89], a framework that includes a sanitizer IPEA-san and a fuzzer IPEA-fuzz. Even if IPEA uses a remote-attestation-based architecture, the proposed systems target different use cases. Although IPEA facilitates the test process for embedded devices, its focus is not on detecting spatial memory safety violations in deployed devices. This difference is reflected in the runtime overhead as IPEA incurs a significantly higher overhead compared to runtime detection tools, as expected from a sanitizer in both general-purpose and embedded contexts.

While exploring the realm of modern runtime remote attestation for general-purpose devices, several solutions have emerged, catering to software complexities beyond simple embedded firmware. Notably, ScaRR [95]

and ReCFA [110] have paved the way for applying runtime remote attestation to services within cloud environments. However, their focus lies solely on monitoring control-flow integrity, disregarding data-flow aspects. On a different front, SgxMonitor [96] presents a pioneering runtime remote attestation protocol explicitly tailored for SGX enclaves. Although SgxMonitor introduces a stateful model, its approach remains confined to globally defined structures. It should be noted that these protocols, similar to those for embedded devices, do not perform vulnerability analysis, underscoring a key challenge that remains open in the field of runtime vulnerability detection of embedded systems.

Vulnerability Analysis While exploring techniques to identify the location of bugs in the presence of anomalies, such as crashes, we identified several options. Notably, sanitizers like AddressSanitizer [87] (ASan), or more recent optimized versions [109] (ASan--), stand out as state-of-the-art tools for monitoring spatial and temporal errors. However, it is important to note that sanitizers are primarily designed for testing purposes, rendering them unsuitable for production environments. They introduce significant performance and memory overhead while lacking protection against evasion techniques or attacks targeting the sanitizer itself. Alternatively, postmortem techniques analyze a core dump of the process to locate errors, bypassing the need for sanitizers. Examples include works such as [102, 37, 101], which may require additional contextual information like process runtime traces from Intel PT, as seen in REPT [37]. Zhenyu et al. introduced Ninja [79], a novel malware sandbox technique built upon ARM TrustZone. The primary objective of Ninja is to improve malware analysis by leveraging ARM TrustZone to effectively counter malware sandbox evasion. This system focuses on bolstering malware analysis capabilities, and is not fit for production environments. Moreover Ninja gathers runtime information by capturing API/syscall invocations, making it incompatible for bare-metal embedded devices.

Value Set Analysis In a specific domain of postmortem analysis, researchers have utilized Value Set Analysis [36] (VSA) to analyze the core dumps. VSA is a static analysis technique that infers variable ranges, aiding in error localization. To enhance the precision of VSA, POMP++ [73] proposes a VSA approach based on reverse execution from the crash point. On the other hand, DEEPVSA [54] addresses aliasing issues using a deep learning model. However, both POMP++ and DEEPVSA rely on specific hardware features, such as Intel PT, and core dumps for analysis. Consequently, they are unsuitable for deployment in embedded systems.

2.2 Vulnerability Testing

Testing has long been a central activity in software engineering, aimed at ensuring correctness, robustness, and maintainability of programs [75]. Traditional approaches include unit testing, which verifies the correctness of individual components; integration testing, which examines the interactions between modules; and system testing, which validates overall functionality [18]. While these methods are effective at uncovering functional defects, they are less suited to identifying security flaws, which often manifest only under rare or adversarial conditions [70].

Security testing, or vulnerability testing, extends classical testing by shifting the focus from expected functionality to resilience against malformed, malicious, or unexpected inputs. A variety of approaches fall under this umbrella, ranging from manual approaches, such as security-oriented code analysis where an analyst manually checks the code to detect bugs and potential vulnerabilities, to dynamic approaches that include symbolic execution and concolic testing [27, 51], which attempt to systematically explore execution paths by solving constraints over inputs.

Among automated approaches to vulnerability testing, fuzzing has emerged as particularly effective due to its scalability and ability to reveal vulnerabilities without requiring full program models.

2.2.1 Fuzz Testing

Fuzz testing, or *fuzzing*, is a family of automatic random testing techniques first proposed in the '80s [72]. This technique is currently one of the most popular automatic bug detection techniques and has helped in discovering multiple vulnerabilities in recent years. The core idea behind a fuzzer is straightforward: feed a program a large number of inputs, monitor its execution, and report to the end user any input that triggered an unexpected behavior, such as crashes, hangs, or unexpected outputs.

Fuzzers can be classified according to different factors. The first is how the fuzzer generates its inputs. This criterion classifies a fuzzer into one of two categories: *generational* that creates the input from scratch using structural information, such as input grammar, and *mutational* that creates new inputs by mutating the previously existing one. The second classifies fuzzers based on the amount of information they have access to from the program under test:

- *Black-box fuzzers* do not have access to the target code and extract information via hardware systems, e.g. Intel PT, or by running the target in an emulator.
- *White-box fuzzers* have access to the whole program and can perform any static analysis beforehand.

- *Grey-box fuzzers* use lightweight binary instrumentation to gain insight into the program at runtime. This represents a trade-off that should result in an improved effectiveness w.r.t. pure black-box approaches, while still avoiding expensive program analysis techniques that could significantly hinder the efficiency of the process.

A typical mutational grey-box fuzzer, which is the focus of both Chapter 5 and Chapter 4, is composed of the following modules:

- **Corpus** A collection of inputs that the fuzzer uses, and mutates, during the fuzzing campaign.
- **Scheduler** A component tasked with selecting the next input to fuzz through some heuristic.
- **Mutators** A set of one or more seed to seed functions that modify the scheduled input.
- **Executor** The component tasked with materially executing the software under test with the mutated input.
- **Feedback** A decision mechanism that, based on information collected by the executor, determines if the seed is valuable and should be retained, for either further mutation or as a crash, or should be discarded.

Among these components the *feedback* showed significant capability in terms of determining the fuzzing exploration behavior. The *standard* feedback relies on code coverage [26, 48, 68, 88, 81]), and leads the fuzzer to only store inputs for further mutation if they discover new edges in the program.

Having introduced the fundamental principles and classifications of fuzzing, we now turn to prior research that has extended these techniques. In particular, we review advances in feedback mechanisms, bug complexity analysis, and context-sensitive fuzzing, as well as recent efforts to adapt fuzzing to embedded systems. These works illustrate both the evolution of fuzzing and the limitations that motivate our contributions.

Fuzzing Feedback Feedback is a very active research area [24, 67, 20, 105] and many papers have shown that modifications to the feedback significantly affect the ability of a fuzzer to overcome roadblocks and its effectiveness at exploring the target program. This means that the feedback component plays a crucial role in steering the fuzzer towards the discovery of *different kinds* of bugs. Recent research has addressed this aspect by incorporating feedback mechanisms that extend beyond edge coverage. Fioraldi et al. [46] introduced state-sensitive feedback using likely invariants to detect violations of expected conditions. Similarly, Liu and Meng [67] proposed DS-Fuzz, a fuzzer that tracks dependent state transitions to uncover deep-state

bugs. Work by Wang et al. [98] leveraged memory access patterns to define *data coverage* as an additional feedback signal, discovering long-hidden bugs in extensively fuzzed projects. Mantovani et al. [69] and Zhou et al. [111] explored path and state sensitivity using data-flow and symbolic execution, respectively, to reach bugs hidden behind complex triggering conditions. These works show that introducing alternative feedback mechanisms significantly improves bug discovery, particularly when execution paths or internal state constraints are non-trivial.

Data-Flow and Context-Sensitive Fuzzing. Recent fuzzers such as GreyOne [50], Beacon [111], and DataFlow [59] have demonstrated the value of incorporating program semantics—particularly data-flow relationships and context sensitivity—into the fuzzing process. By moving beyond syntactic coverage metrics like edge or block coverage, these tools leverage richer information about variable dependencies, interprocedural flows, and calling contexts to guide the generation and selection of inputs. This enables more precise exploration of program behaviors, especially in scenarios where the vulnerability is conditioned on complex interactions across functions or data paths.

For instance, GreyOne [50] uses taint tracking to guide fuzzing toward deeper data-dependent conditions; Beacon [111] employs selective context cloning to prioritize relevant execution contexts; and DataFlow [59] uses dependency graphs to focus on inputs affecting sensitive operations. These approaches have proven effective in uncovering bugs that are unreachable or invisible to conventional fuzzers, especially those hidden behind opaque state transitions or guarded by non-trivial control logic.

However, while these techniques significantly improve bug detection in targeted domains—such as error-handling code, pointer dereferencing, or user-influenced computations—they are generally designed with specific classes of vulnerabilities in mind. Consequently, they provide fuzzing enhancements rather than general diagnostic tools for understanding *why* certain bugs that are harder to detect than others.

In contrast, our work aims to provide a broader and more systematic perspective. Rather than proposing new feedback signals or instrumentation strategies, we focus on analyzing and quantifying the *complexity* of individual bugs. Our methodology is agnostic to the fuzzer’s internal mechanisms and can be applied to a wide range of programs and vulnerabilities. By decomposing bug complexity into three orthogonal components—reachability, path constraints, and state conditions—we offer a general-purpose framework for evaluating the intrinsic difficulty of triggering a vulnerability.

Fuzzing in Embedded Systems Fuzzing techniques for embedded devices can be broadly classified into two major categories: hardware-in-the-

loop approaches and emulation-based approaches.

Emulation-Based Approaches. Emulation-based fuzzing techniques aim to execute firmware in a simulated environment, often without requiring access to the physical hardware. This approach enables scalable testing while avoiding hardware constraints, making it particularly useful for analyzing a wide range of embedded systems. Several notable works fall into this category. DICE [71], Fuzzware [84], P2IM [45], and Laelaps [28] focus on peripheral modeling and inference. These frameworks leverage techniques such as symbolic execution, peripheral abstraction, and model learning to approximate the behavior of real hardware peripherals. This enables effective firmware execution in an emulated environment, improving the efficiency of fuzz testing. Other approaches focus on rehosting firmware in a way that abstracts the underlying hardware while preserving its execution semantics. HALucinator [35] targets the hardware extraction layer (HAL) by intercepting HAL calls and providing simulated or stubbed responses to maintain the firmware execution flow. Pretender [55], on the other hand, models Memory-Mapped I/O (MMIO) peripherals by dynamically observing and learning from the behavior of the original hardware. These techniques allow fuzzing frameworks to execute firmware more accurately in an emulated environment, while reducing the reliance on manually crafted peripheral models. Despite their advantages, emulation-based approaches often face challenges in accurately reproducing real-world execution environments, particularly when dealing with complex or undocumented peripherals. The inability to perfectly replicate hardware interactions can sometimes lead to discrepancies between the observed and actual firmware behavior.

Hardware-in-the-Loop Approaches. In contrast to emulation, hardware-in-the-loop (HIL) fuzzing techniques execute firmware directly on the target device, allowing more accurate testing by interacting with actual hardware components. IPEAFuzz [89] introduces a lightweight instrumentation method that facilitates data exchange between the target device and the fuzzer, allowing efficient real-time monitoring and feedback collection. μ AFL [31] employs ARM-specific debugging mechanisms, such as hardware failure points and trace features, to enable fuzzing without requiring modifications to the firmware itself. This approach minimizes interference while preserving the fidelity of the testing process. HIL approaches provide high-fidelity and realistic testing conditions, but suffer from hardware dependency and scalability issues. The requirement for physical devices imposes constraints on execution speed, automation, and reproducibility, making large-scale or parallel testing more challenging compared to emulation-based methods.

Both hardware-in-the-loop and emulation-based approaches play crucial roles in fuzzing embedded systems, each with its own trade-offs. While emulation enables scalability and flexibility, it struggles with hardware fidelity. In contrast, hardware-in-the-loop methods offer high realism but face challenges related to scalability and accessibility. Recent advances, such as

hybrid approaches that combine elements of both techniques, are emerging to bridge these gaps, offering promising directions for future research in embedded firmware security testing. In our work, we present a novel fuzzing framework that is leveraged to improve scalability and speed while maintaining high fidelity in firmware testing. Our approach outperforms existing methods by making it well suited for large-scale embedded system security testing.

While fuzzing provides an effective mechanism for exploring vulnerabilities, not all bugs are equally easy to trigger. To fully understand why some flaws elude detection despite extensive testing, we now turn to the concept of bug complexity.

2.2.2 Bug Complexity

Software bugs differ not only in their manifestation but also in the difficulty of discovering and reproducing them. Some can be reliably triggered with simple inputs, while others require intricate sequences of conditions or interactions between components. This notion, broadly referred to as *bug complexity*, captures the inherent difficulty of activating a vulnerability. Understanding bug complexity is crucial for two reasons: it shapes how effective testing techniques are at uncovering flaws, and it provides valuable insights for building more realistic benchmarks and evaluating bug-detection tools. In the following, we first introduce classical classifications of bugs and their historical foundations, and then connect these perspectives to software complexity metrics and modern fuzzing research.

Bug Complexity and Classification. The idea that bugs differ in complexity has a long history. Grottke et al. [53] introduced a seminal classification distinguishing between *Bohrbugs*, which are deterministic, consistently reproducible, and relatively straightforward to detect, and *Mandelbugs*, which are non-deterministic, sporadic, and often difficult to reproduce due to their dependence on subtle program states or rare runtime conditions. This taxonomy emphasizes that not all faults are created equal: some can be reliably identified and patched through conventional testing, while others remain elusive due to the complexity of their activation conditions.

Building on this foundation, Ratliff et al. [82] empirically demonstrated that Mandelbugs tend to involve a higher number of interacting components, such as variables, input dimensions, or environmental factors, indicating that bugs requiring complex interaction patterns are inherently more difficult to trigger. Their findings support the hypothesis that bug complexity correlates strongly with the level of internal and external conditions necessary for activation.

A complementary body of work has attempted to model bug complexity indirectly through Software Reliability Growth Models (SRGMs) [63, 62, 91,

64]. These approaches treat the delay between fault detection and resolution as a proxy for complexity and typically classify bugs as simple, hard, or complex based on the effort or time required to fix them. While useful in large-scale empirical settings, these models operate on aggregated metrics and are less suited for detailed analysis of individual bugs.

In contrast to both abstract taxonomies and statistical reliability models, our work provides a concrete, fine-grained methodology for empirically measuring the complexity of individual bugs. We do not infer complexity post hoc from repair timelines or bug behavior during debugging. Instead, we operationalize the Bohrbug–Mandelbug spectrum through a fuzzing-based framework that quantifies triggering difficulty based on three orthogonal dimensions: *reachability*, *path constraints*, and *state conditions*. This allows us to assess, under controlled and replicable conditions, which bugs are hard to discover and why—offering valuable insight for both fuzzer evaluation and benchmark construction.

Fuzzing and Software Complexity. The notion of bug complexity is crucial when discussing automatic vulnerability detection techniques. Understanding which classes of bugs a technique can expose helps to identify possible biases in the analysis process and can guide the development of tools towards finding harder-to-detect flaws. However, a key limitation of bug complexity as traditionally defined is that it is inherently *descriptive*: it characterizes bugs after they are discovered, but does not provide actionable guidance during testing. By contrast, testing techniques often require *predictive* metrics to steer the exploration process effectively. In this context, software complexity has emerged as a practical proxy.

Fuzzing research has investigated the relationship between software complexity and the likelihood of discovering bugs. Zhang et al. [107] demonstrated that static code metrics, such as function size, nesting depth, and control-flow complexity, correlate with fuzzing performance across various programs and fuzzers, suggesting that such features can be used to predict areas of code that are harder for fuzzers to explore effectively. Classical studies [76, 30] similarly established that complexity indicators like code churn, cyclomatic complexity, and change frequency are strong predictors of defect density and software fragility.

These works highlight a consistent theme: more complex code is not only more error-prone but also harder to analyze and test automatically. While these efforts typically focus on identifying “bug-prone” regions in code or predicting where bugs may reside, there is a lack of focus in analyzing the complexity *after* a vulnerability has already been discovered.

Chapter 3

Efficient runtime detection of spatial memory errors in embedded systems

3.1 Problem Statement

Embedded systems have unique characteristics such as limited resources, restrictions on updates, and real-time constraints, which complicate the application of traditional attack detection methods. In particular, IoT firmware may suffer from memory errors that lead to data-only and control-flow attacks [83] which can have severe consequences such as crashes, data breaches, or even physical damage. Previous efforts to design detection systems to address memory errors in the embedded systems landscape have typically relied on remote attestation methods, incorporating techniques such as Control-Flow Integrity (CFI) [15, 93, 103]. However, these techniques lack the ability to perform *vulnerability analysis*, which is essential to identify vulnerabilities and prevent future intrusions. Crash reporting (e.g., vulnerability analysis) is crucial yet challenging in embedded systems due to the necessity of identifying and addressing vulnerabilities in a constrained environment where traditional debugging tools and methodologies are often limited or impractical. In contrast, in less constrained environments, such as desktop PCs, many products routinely collect and report crash data to vendors. This practice, exemplified by Mozilla and Ubuntu with their crash reporting systems [2, 8], aims to improve the quality and security of their products.

Building upon the goal of designing a crash reporting system for embedded devices, this chapter presents EMBEDWATCH, a groundbreaking system that merges fat pointer principles with remote attestation, tailored specifically for embedded devices. EMBEDWATCH distinguishes itself by its ability to effectively enforce the spatial memory safety, performing a comprehensive analysis of various memory segments including stack, heap, and global vari-

ables. The key innovation of EMBEDWATCH lies in its vulnerability analysis capability, which accurately identifies the exact code segment responsible for a bug, enhancing precision in error detection and resolution. Unlike conventional approaches that are reliant on Control Flow Integrity (CFI) systems [15, 93, 103], EMBEDWATCH introduces a broader and more comprehensive strategy. This system extends beyond merely detecting control-flow hijacking vulnerabilities; it adeptly identifies and locates vulnerabilities associated with control-flow and data attacks. Consequently, EMBEDWATCH equips IoT devices with the capability to autonomously generate and send detailed crash reports of such attacks, improving their security and response mechanisms.

Our framework presents an innovative static framework called a data graph model that utilizes node selection and runtime data to identify Out-of-Bound (OOB) memory errors, including attacks to data structure parameters. *It is important to note that we exclude spatial memory errors that do not lead to successful attacks, such as scenarios where the overflow cannot be manipulated by the input program.* This model not only pinpoints critical locations in the code prone to OOB errors, but also guarantees their thorough monitoring during execution. The data graph model merges flawlessly with the existing SVF [92] framework, boosting its capabilities for accurate interprocedural analysis and firmware code instrumentation. The instrumented firmware is specifically configured to monitor memory structures that are more susceptible to attacks, such as buffers, thus effectively minimizing run-time overhead. To further optimize performance and reduce this overhead, we have implemented additional design optimizations based on the cache concept that enhance efficiency without compromising security. During runtime, with the support of a trusted anchor, such as ARM TrustZone [60], the device securely transmits a minimal, yet highly reliable, data trace that represents the current running status of the program represented by a data graph model. This approach not only secures the data in transit, but also ensures that the information is reflective of the real-time execution state of the firmware. Upon reaching the IoT backend, these data undergo a thorough processing phase. Here, the system analyzes the information to detect any instances of spatial memory violations. The backend is equipped with an algorithm capable of dissecting and understanding the details of these violations. As a result, the IoT backend generates detailed crash reports that pinpoint the vulnerability’s origin in the source code.

EMBEDWATCH has been developed to enable firmware vendors to produce software that communicates the execution status in real time, helping to quickly deploy patches. Consequently, we argue that it is crucial to have access to the firmware’s source code. This access reveals detailed information, including the position of the memory structures, the precise sizes of each memory object, and the techniques employed in memory management allocation.

To assess EMBEDWATCH, we conducted a set of experiments that measure runtime overhead and stress the security guarantees. Specifically, we deploy EMBEDWATCH in real-world firmware containing genuine CWEs [32]. Our results show that EMBEDWATCH exhibits remarkable precision in identifying the underlying source of memory errors within the analyzed firmware. Additionally, we observe a runtime overhead range between 0.01% and 2.33%.

In summary, this chapter makes the following contributions.

- We present EMBEDWATCH, a groundbreaking crash reporting system that employs a fat pointer technique for embedded devices in production environments. We have created, implemented, and successfully tested EMBEDWATCH, which accurately detects vulnerabilities in spatial memory attacks, thus enhancing IoT backends (section 3.4). The source code for EMBEDWATCH is open source and can be accessed at [4].
- We introduce a static framework named the data graph model (section 3.4), which examines firmware code by identifying nodes and runtime data that can detect Out-of-Bound (OOB) memory errors. This data graph model integrates effortlessly with the current SVF framework [92], improving its precision in interprocedural analysis. Moreover, our framework incorporates optimization techniques to minimize performance overhead and to address data attacks on memory structure fields.
- We evaluated EMBEDWATCH on real-world embedded programs that cover broad scenarios of use of IoT devices, demonstrating the efficacy and practicality of EMBEDWATCH in real-world cases. Our experimental results with in place optimizations demonstrate that EMBEDWATCH reduces the initial overhead from (0.07% - 36.4%) to (0.01% - 2.33%).

3.2 Threat Model

Our system trusts the code running inside the Secure World (*e.g.*, the measurement engine) and assumes that attackers cannot bypass TrustZone protection. We also trust our compiler and the trampoline code, and assume that attackers cannot inject code into the Normal World or tamper with the instrumented code or the trampoline library. This can be enforced using code integrity protection methods for embedded devices [34, 65], which are not the focus of this chapter. In the Normal World, attackers may exploit spatial memory error vulnerabilities [94] in embedded systems to launch various types of memory error attacks, such as Return-Oriented Programming (ROP) and Data-Oriented Programming (DOP) attacks. These attacks can compromise the control flow of the firmware application and access criti-

cal data from an embedded program. Additionally, attackers may abuse unprotected interfaces of the embedded program to force the device to perform unauthorized or unintentional operations. To address these issues, our system is designed to detect these types of attacks against the system (subsection 3.5.3). Generally, our system adheres to the defensive strategies employed by fat-pointer systems.

3.3 EMBEDWATCH Overview

Our system offers a variety of benefits compared to classic vulnerability analyzers [102, 73, 101], such as (1) eliminating the requirement of large and unnecessary core dumps, (2) reducing the amount of traced information created by the instruction execution trace [7], and (3) increasing resistance against attempts to circumvent the system by attackers. To provide an understanding of EMBEDWATCH, we present a practical example of vulnerability analysis in embedded systems (subsection 3.3.1). Following that, we describe the system architecture in detail (subsection 3.3.2).

3.3.1 Running Example

In Listing 1, we present the code snippet of the *Heat Press* firmware, extracted from our dataset, designed to control a press machine. The CWE vulnerability in question pertains to the out-of-bounds type and is specifically found within the Modbus library used for communication with peripheral devices.

In particular, the function `Modbus_poll` utilizes the `fgetc` function to read characters from the standard input (`stdin`) (Line 21). These characters are then stored in a buffer named `au8regs`. The firmware reads exactly one hundred characters and subsequently calls the `Modbus_get_FC3` function (Line 13), passing the `regs` buffer as an argument. The issue arises from the fact that the `Modbus_get_FC3` function lacks proper boundary checks. Consequently, if an attacker overflows the size of the `regs` buffer, adjacent memory locations can become corrupted, leading to potential security vulnerabilities.

To effectively identify out-of-bounds (OOB) vulnerabilities and provide meaningful crash reports, an analyst requires ample contextual information. This includes details such as (1) the specific object involved in the memory operation (e.g. memory structures, buffers), (2) the boundaries of the object, and (3) the stack trace at the time of the OOB occurrence.

Looking back at the example in Listing 1, we notice that the function `Modbus_get_FC3` operates on a buffer that could potentially be a parameter of the function call itself. The challenge here is that during the static analysis stage, we cannot determine which execution path the program will take,

and consequently, we do not know which function parameters will be considered in case of multiple calls of the same function with different buffers. This lack of information during static analysis could lead to ambiguity in identifying the correct object involved in the out-of-bound (OOB) issue. However, EMBEDWATCH effectively resolves this ambiguity by generating a reliable trace that accurately tracks the progression of key variables in terms of their liveness. To achieve this, our instrumentation system records the base address of the relevant object, such as `au16regs` in the example (line 3), by leveraging run-time information, such as definition and the executed calls. By doing so, EMBEDWATCH provides the necessary context to identify the function’s parameters and accurately pinpoint the location of the OOB vulnerability. For instance, when `Modbus_get_FC3` is called, our instrumentation system identifies that `au16regs` is being used as the `regs` parameter for `Modbus_get_FC3`. This crucial information reveals the actual object under consideration and allows us to infer the boundary of `regs` (i.e., based on the base address and size of `au16regs`). By understanding the reference of the variable `regs` and the addresses present in the loop a remote agent can determine whether the firmware works within the correct limits. Specifically, it can verify whether the value of `regs[i]` resides within the memory range defined by `au16regs`.

Upon detecting an out-of-bound (OOB) issue, a remote agent possesses the capability to generate a crash report. This comprehensive report includes details on the location of the OOB (line 13) and the object involved (e.g. `au16regs`, as defined in line 3). By tracing the sequence of function calls, particularly in our example, the `Modbus_get_FC3` and `Modbus_Poll` calls, in terms of the stack frame, the remote agent can effectively track the sequence of events leading up to the point of vulnerability. Within this context, three crucial challenges must be addressed:

- (C1) A technique consists of detecting erroneous memory accesses through the use of the memory state (subsection 3.4.1).
- (C2) A remote attestation technique for remotely obtaining a concise representation of the memory state of the embedded device (subsection 3.4.2, subsection 3.4.3, and subsection 3.4.4).
- (C3) A vulnerability analysis utilizes the inferred memory state and out-of-bound access to determine the bug’s location (subsection 3.4.5).

3.3.2 System Architecture

Figure 3.1 depicts the high-level architecture of EMBEDWATCH, which consists of two main components: *Offline Preparation* and *Online Verification*.

```

1  #include <stdio>
2  uint8_t au8Buffer[100];
3  uint16_t au16regs[20];
4
5  void Modbus_get_FC3(uint16_t *regs)
6  {
7      uint8_t u8byte, i;
8      u8byte = 3;
9
10     for (i=0; i< au8Buffer[ 2 ] /2; i++) {
11         regs[ i ] = word(
12             au8Buffer[ u8byte ],
13             au8Buffer[ u8byte +1 ] );
14         u8byte += 2;
15     }
16 }
17
18 void Modbus_poll(){
19
20     for(int i = 0; i < 100; i++)
21         au8Buffer[i] = fgetc(stdin);
22
23     Modbus_get_FC3(au16regs);
24     u8state = 1;
25 }

```

Listing 1: Code snippet showcasing a CWE vulnerability in the Heat Press firmware, where the Modbus library is susceptible to out-of-bound access.

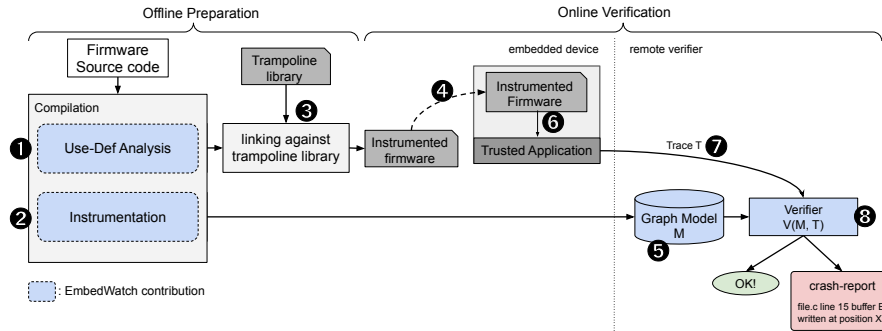


Figure 3.1: EMBEDWATCH Architecture. During Off-line Preparation, the firmware source code is compiled and analyzed. In this phase, we obtain an instrumented firmware and a model that describe the data-flow relationship between inputs and internal buffers. During the online verification, the instrumented firmware is executed on a embedded device and reports a trace T to a Verifier. The latter validates the trace T against the model M. In case of violation, the Verifier reports the incident

In the *Offline Preparation* phase, we conduct program analysis techniques on the source code (❶) and instrument the firmware to trace runtime information (❷). Subsequently, we link the instrumented firmware with a trampoline library to facilitate communication with the trust application (❸), and install it in the embedded device (❹). The compilation process also generates a graph model M (❺), outlining the usage of critical buffers in the program. This model allows a remote agent to reconstruct the firmware’s memory state, which aids in the effective detection of Out-of-Bound errors and vulnerability analysis.

During the *Online Verification* phase, the device communicates with the TrustZone (❻), transmitting a runtime trace T to the Verifier (❼). Finally, the remote Verifier uses the model M to identify potential out-of-bounds accesses and generates root cause reports based on trace T (❽). In the following section, we provide an overview of the software components and algorithms utilized by our system.

3.4 EMBEDWATCH Protocol

EMBEDWATCH is the first work that merges fat pointer principles with remote attestation, specifically tailored for embedded devices. As such, one of the major challenges is the ability to keep overhead low while effectively deducing the program variables utilized during the firmware’s operational phase. To address this, our system monitors dynamic information such as: the objects that are active, determines their boundaries, and reconstructs the call stack when an out-of-bounds (OOB) error occurs. In the following sections, we detail how EMBEDWATCH achieves this goal by showing the protocol in subsection 3.4.1. We expand the example by introducing the graph data modeling in subsection 3.4.2, which allows us to find crucial instrumentation points. Then, we describe how the firmware collects runtime information in subsection 3.4.3 and an optimization to reduce overhead in loops in subsection 3.4.4. Finally, we show how EMBEDWATCH correctly detects and reports OOB errors in subsection 3.4.5.

3.4.1 The Protocol in a Glance

EMBEDWATCH implements a remote attestation protocol between two points: the Prover, an inline agent defined within the firmware, and the Verifier, an external application aimed at identifying memory violations. To trace runtime information in the Prover, EMBEDWATCH uses instrumentation points provided by our data graph model. Without loss of generality, the protocol considers two main types of packets: *Definition*, which indicates the instantiation of new variables (e.g. alive variables) and *Use*, which represents a memory operation. During compilation time, our graph model assigns a unique ID for each *Definition* and *Use* of the variable. This ID is used

to uniquely identify the information of the variables (e.g., memory boundaries) during the execution path and locate their corresponding definitions in a static view of the firmware. Figure 3.2 illustrates a simplified version of the protocol that only considers a single definition and the use of the same variable. The *Definition* D_b is related with the variable $\mathbf{b}$, whose size is 10 bytes (❶). The size information is retrieved by the static components of our framework and combined with the runtime base address from the incoming *Definition* and stored in the variables alive table (❷). At this point, the Verifier is aware that the buffer $\mathbf{b}$ is instantiated at the address `0xFF03BA`. Then, for each *Use*, the Verifier inspects the model to infer the involved buffer and validate the memory access. In Figure 3.2, the *Use* U_b is the only use connected to *Definition* D_b . Therefore, the Verifier infers that U_b operates on the variable b . This allows the Verifier to decide whether the memory access falls within the variable range (❸). In case *Use* accesses outside the buffer, an out-of-bound is detected (❹). The operation *Use* reveals the location outside the bound in the source code, while the variable involved provides additional contextual information about the location of the code bug location (❺). The complete model is detailed in subsection 3.4.2. At runtime, our protocol collects additional information to handle more complex cases such as aliasing and `struct` fields.

3.4.2 Graph Data Modeling

The core mechanism of EMBEDWATCH revolves around a data graph model that represents connections between variables and their use in the firmware code, represented by the Graph Model M (❺ in Figure 3.1). This model represents our contribution and is utilized for the detection of Out-of-Bound errors and vulnerability analysis, enabling the Verifier to trace the lifetime of variables and determine their boundaries. Our primary goal is to perform a use-def analysis on the memory objects defined in the firmware code, which necessitates the creation of a value flow graph (VFG). The purpose of a VFG is to model the flow of data values through a program, allowing the analysis and tracking of how values are assigned, propagated, and transformed across different variables and point of the program. To construct the VFG, we use the SVF framework [92], which builds an interprocedural Memory SSA form that captures definition use chains for both top-level and address-taken variables, connecting them with value flow edges. The interprocedural static value flow analysis framework (SVF), built on LLVM, performs precise static value flow analysis of C/C++ programs. It integrates Andersen’s pointer analysis to understand pointer usage and dereferencing.

Our framework extends the SVF tool [92] to develop our data graph model as follows. Once the VFG is constructed, our static framework navigates the graph, recognizes the nodes based on their roles, and labels each node with the correct ID along with dynamic information that will be sent

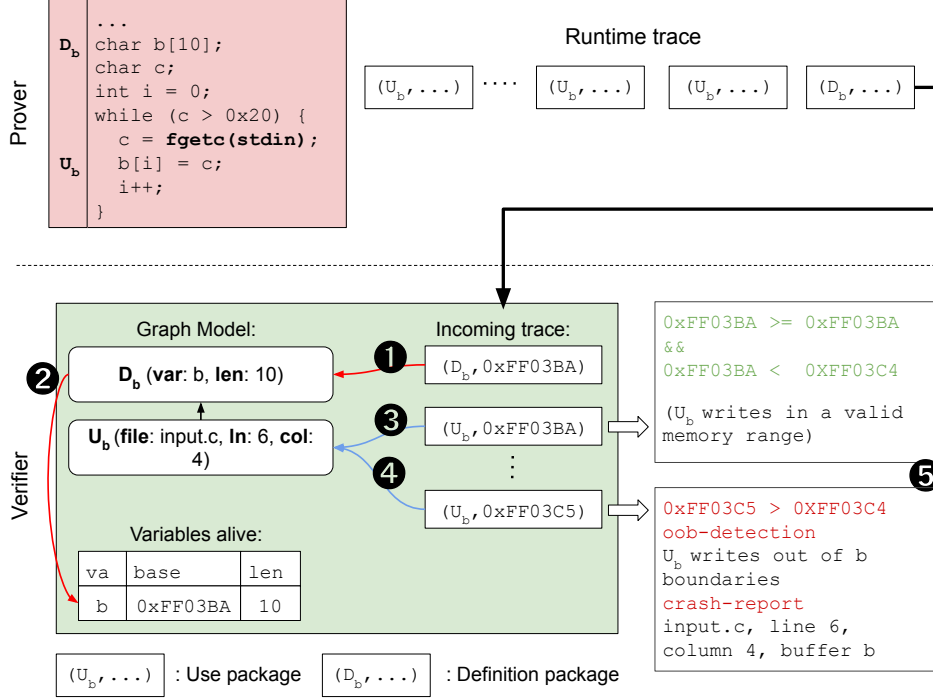


Figure 3.2: EMBEDWATCH Protocol. The prover sends a stream of **Use** and **Definitons** that represents the variable alive and their use. The Verifier infers the firmware state from the runtime trace and the Use-Def graph. The Use-Def graph reduces the size of the runtime trace in two ways (1) it avoids transmitting static information (*e.g.*, the buffer **b** length), and (2) it minimizes the instrumentation points (the only possible out-of-bound use is U_b).

to an external Verifier. Specifically, we identify three types of nodes in our model: *Input functions* (IN), *Use Nodes* (USE), and *Definition Nodes* (DEF). These nodes hold various pieces of information and aid in assessing the firmware’s integrity status.

Input Nodes To detect OOB accesses, EMBEDWATCH locates the input functions (IN) in the firmware code. These functions act as attack entry points, allowing adversaries to corrupt the firmware’s internal status through OOB accesses. We need to distinguish two types of input functions based on their semantics. SIN and PIN. SIN functions return scalars (*e.g.*, `getc`), often used in loops to transfer single bytes to buffers (*e.g.*, in Listing 1). For SIN functions, we trace all written memory locations. In contrast, PIN functions directly fill buffers (*e.g.*, `scanf`) and are required to return the number of read bytes, which we use to calculate and trace the last written

address. Our prototype encodes five popular IN functions (*e.g.*, `gets`, `getc`, `fgets`, `fread`, and `scanf`), which were sufficient to handle all firmware in our experiments. However, our prototype can be extended to any function that fulfills our IN function definition, such as `uart_read` [12].

Use Nodes Once we locate the IN functions, we proceed with a context-sensitive analysis based on the forward process in the interprocedural VFG [92] to detect all the possible corresponding Use nodes (USE).

There are two types of USE nodes, similar to IN functions: SUSE and PUSE. SUSE nodes store scalars in memory regions, such as “STR R2, [R1]” in the ARM assembly, and we trace the written memory location for these nodes. PUSE nodes manipulate buffer pointers, such as `memcpy`. We require PUSEs to return the number of bytes written, just as we do with PIN, so that we can calculate and report only the last modified address. We need to distinguish SIN/SUSE from PIN/PUSE because the former are implemented as simple memory transfers (*e.g.*, STR), from which we can easily report each address written. For the latter, instead, we only observe the amount of memory transferred (*e.g.*, `memcpy`), thus we need to calculate the last written address. For this purpose, we design a specific instrumentation in subsection 3.4.3.

Def Nodes DEF nodes provide critical information about object boundaries, which can be located in various memory regions such as the stack, global sections, or heap, each with unique characteristics. For our framework, it is essential to accurately identify variable definitions along with their memory boundaries to infer out-of-bound errors (OOB) and produce meaningful crash reports. Specifically, our framework handles four types of DEF nodes: SDEF for stack variables, GDEF for global variables, HDEF for heap variables, and FDEF for fields in `structs`. For stack buffers (SDEF), which can be allocated across multiple stack frames, we further associate an index referring to the specific stack frame in which they are allocated. This information is useful for identifying the execution path that triggers the attack. For SDEF and GDEF, their sizes are extracted at compilation time. HDEF refers to `malloc` allocations, with sizes computed at runtime. FDEF allows us to infer field boundaries and detect overflows within a `struct`. This data is obtained by analyzing the function data flow: for each DEF node with a *struct type*, we reconstruct all accessed fields between the definition and the corresponding use. To achieve this, we utilize LLVM’s GEP (GetElementPtr) instruction [11], which provides functionalities for accessing struct fields. By examining GEP instructions, we can precisely identify the memory structures and their fields in use, thereby determining their sizes accurately. After extracting the sizes of the definition nodes, we also need to trace their memory addresses at runtime to perform out-of-bounds

(OOB) checks. Therefore, at runtime, we trace the base addresses of buffers for [G, H, S]DEF. Combining these base addresses with the buffer sizes allows us to calculate the buffer boundaries. FDEF is not traced at runtime but is included in the model M for trace verification purposes (details in subsection 3.4.5).

Linking Nodes Procedure After determining all node types, sizes, and base addresses, it is necessary to associate the appropriate uses (USE nodes) of variables with their definitions (DEF nodes). In the results of the application of forward-based context-aware analysis within the interprocedural VFG [92], multiple definitions of the same variable can be associated with a single use. For example, this situation arises when a function is called with parameters that are defined at different points in the code. Unfortunately, the specific definition that will be triggered at run-time cannot be statically determined due to the unpredictability of the program’s execution path. To solve this problem, we initially identify all potential DEFs that the use can target as candidates. Since each entity within our analysis framework is assigned a unique ID, we can easily differentiate them. At the use point, our framework gathers not only the data pertaining to the use being executed, but also the ID of the definition targeted by the operation.

Moreover, when memory objects are passed as function arguments, our framework needs to propagate the IDs of the corresponding definitions for inter-procedural analysis. This is achieved by automatically adding an extra ID argument to each analyzed function and modifying all corresponding call sites to propagate the runtime information accordingly.

3.4.3 Firmware Instrumentation

EMBEDWATCH utilizes source code instrumentation to track USE and DEF operations, ensuring the Verifier can accurately detect out-of-bounds (OOB) accesses and produce reliable crash report even in the presence of potential attackers (as outlined in section 3.2).

For every Definition (DEF) node, we monitor the buffer ID and its base address. Global Definition (GDEF) nodes are tracked during the process’s initial bootstrapping phase and are logged only once. In contrast, Stack Definition (SDEF) and Heap Definition (HDEF) nodes are monitored after variable declarations by adding instrumentation points at the allocation sites and linking them to the variables that reference the allocated memory regions. This provides the Verifier with information about the buffer’s location in memory. The buffer size is inferred from the source code for static allocations (*i.e.*, stack or global) or traced by instrumenting the argument of `malloc`-like functions for heap allocations. For each Use (USE) instruction, we record the instruction ID and the written address. In case of PIN/PUSE nodes, we calculate the last written address by adding the ini-

tial buffer pointer and the number of bytes written. When memory objects are passed as function arguments, EMBEDWATCH passes along the buffer IDs and their stack index as additional arguments. This is achieved by automatically adding extra buffer ID/stack index arguments to each analyzed function and modifying all corresponding call sites to propagate the runtime information. Moreover, we track the stack frames to differentiate between the local buffers of the functions. Specifically, the embedded device keeps a stack index that we increase and decrease at each function’s prologue and epilogue, respectively. We then add the stack index as an additional field along with the other model nodes. Having the stack index helps the Verifier determine which stack frame the firmware is currently executing. To avoid tampering, we placed the stack index in a shadow area, as described further in subsection 3.5.3.

3.4.4 Loop Optimization

During our experiments, we observed that certain loops caused significant runtime overhead, especially in scenarios involving large communications with the TrustZone. This was primarily due to the high number of memory operations performed by the firmware. To solve this problem, we store the USE/DEF nodes occurring within specific loops in a buffer in the Normal World. Once the loop (or the firmware) ends or when the buffer is full, we transfer the stored nodes in bulk to the Secure World. However, this optimization is only applied to loops that do not involve function calls, indirect jumps, or return instructions, as these could enable attackers to evade our system by skipping the synchronization operations with the TrustZone. In contrast, for loops without control-flow transfers, adversaries can only activate the attack payload once the loop itself terminates. This optimization approach is effective and secure for two reasons. First, tampering with the cache can only occur when the payload is activated, which happens after the loop ends and the cache has already been flushed into the TrustZone. Second, the cache is stored in a randomly allocated shadow area, similar to SafeStack [10], making it difficult for adversaries to identify its location. In our experiments, the cache mechanism significantly improved performance, yielding approximately 13.4 times better results compared to standard instrumentation (subsection 3.5.2). More details on the security analysis of this solution are presented in (subsection 3.5.3).

3.4.5 Trace Validation

The Verifier (represented by ⑧ in Figure 3.1) validates the USE/DEF nodes reported by the firmware. Its purpose is to infer the firmware’s memory status, detect OOBs, and produce crash reports in case of errors. The Verifier is composed of three main components: (1) one for tracking the

Algorithm 3.4.1: Variable Alive Update

Input: Model M
Input: Runtime Node d

```
1 stack_index ← d.stack_index;
2 node_type ← d.node_type;
3 if d is SDEF ∨ d is GDEF then
4   size ← get_size(M, d.id);
5   var_name ← get_var_name(M, d.id);
6   base_address ← d.base_address;
7 else if d is HDEF then
8   size ← d.size;
9   var_name ← get_var_name(M, d.id);
10  base_address ← d.base_address;
11 end if
12 if table_has(stack_index, var_name) then
13   delete_variables(stack_index, var_name);
14 end if
15 insert_table(stack_index, node_type, var_name, base_address, size);
```

firmware run-time status, (2) one for identifying OOBs, and (3) one for producing a crash report. We provide more details in the following sections.

Firmware Execution Memory The Verifier maintains a table of active variables in the firmware based on the information received by the execution trace. In particular, for each variable, it records the following information: the stack index, node type, variable name, size, and base address. The stack index refers to the actual stack frame in which a node is executed. To minimize traced nodes and keep the table information correct, the Verifier infers the allocation of new stack frames from incoming USE/DEF traces without tracing context switch events (such as ret/call from functions). To correctly identify the new stack frame, the Verifier checks for the presence of a new DEF of a local stack variable in the trace and, if found, assumes that the previous frame is no longer in scope and removes its information from the table, reducing the amount of stored data. This policy is implemented in the algorithm line 7. After obtaining the variable name, node type, base address, size, and stack depth from the trace, the system updates the table (see line 15). This approach ensures that the information on the firmware variables is always accurate and up-to-date.

OOB Detection The Verifier can infer the boundaries of objects involved in USE and IN operations and detect Out-Of-Bounds (OOB) by using the alive variables table. The algorithm is described in Algorithm 3.4.2. For each run-time node u , the Verifier obtains its stack index and the written address (line 4-6). If u contains a DEF_ID and DEF_index, it means that the node is operating on a pointer passed as a reference. In that case, the

Algorithm 3.4.2: OOB Detection

Input: Model M
Input: Runtime Node u

```
1 stack_index ← u.stack_index;
2 address ← u.address;
3 if u has [DEF_ID, DEF_index] then
4   d ← get_defintion(M, u.DEF_ID);
5   var_name ← d.var_name;
6   var_stack_index ← u.DEF_index;
7 else
8   d ← get_defintion(M, u.id);
9   var_name ← d.var_name;
10  var_stack_index ← stack_index;
11 end if
12 var_base_address ← get_var_base_address(M, var_name, var_stack_index);
13 var_size ← get_var_size(M, var_name);
14 if oob_detected(store_address, var_base_address, var_size) then
15   vulnerability_analysis(var_stack_index, stack_index, u, d);
16 end if
```

Verifier extracts the buffer definition, its variable name, and stack index from the node *u* itself (line 4 and 6). If not, the buffer definition and the variable name are obtained from the Use-Def graph (line 8), while the stack index remains the same as the incoming node *u* (line 10). Once the variable name and its stack frame have been inferred, the Verifier checks if the address falls within the buffer boundaries by enquiring the table of alive variables. The function `oob_detected` also detects OOB in `struct` fields. If *d* refers to a `struct`, the Verifier searches for FDEFs between the DEF associated with `var_name/var_stack_index` and *u* in the Use-Def graph. For each traversed FDEF, the Verifier can infer the field or subfield that the firmware is pointing to, and compute its boundary. Finally, if an OOB is detected, the Verifier performs a vulnerability analysis (line 15).

Crash Report Once the Verifier detects an OOB, the function `vulnerability_analysis()` produces a report on the anomaly (line 15). In particular, the Verifier reports: (1) the line of code where the OOB occurred, (2) the object involved in the OOB (*i.e.*, buffer or `structs` field) and (3) the stack trace. The stack trace is reconstructed from the table of living variables. In particular, knowing the stack frame where a variable has been allocated (*i.e.*, `var_stack_index`), and the stack frame of the USE node (`stack_index`), allow us to infer the last functions alive in the firmware by inspecting the variables alive table. This reveals contextual information to assess and patch the observed bug.

3.5 Evaluation

We assess the properties of EMBEDWATCH over *three* main research questions: **(RQ1)** Does EMBEDWATCH produce correct vulnerability analysis reports (Efficacy Analysis) (subsection 3.5.1)? **(RQ2)** Can EMBEDWATCH be deployed in real platforms (Performance Evaluation) (subsection 3.5.2)? **(RQ3)** What are the security property of EMBEDWATCH (subsection 3.5.3)?

Prototype Details All experiments are carried out on an ARM Pi 3b + commercial board, which is equipped with ARM TrustZone support and compatible with the OP-TEE commit `10b7b60b`. The selected board mounts a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit processor @ 1.4GHz, and 1GB DDR2 RAM. We selected this board as our reference device for prototyping because it is a widely used IoT development board. The Trusted OS (OPTEE-OS) runs in Secure-Mode (SEL-1), the rich OS (BusyBox) is shipped with OP-TEE. For the compilation module and model generation, we extend the LLVM version 13 and use the SVF commit `cc3151c0`.

Firmware Dataset We assess the EMBEDWATCH system by evaluating a total of 15 open source embedded programs [3]. Among these, two firmwares were sourced from the OAT dataset [93], which comprises a total of 5 firmwares that were used to test the OAT framework. We omit three firmware from the OAT dataset due to the manual effort required to port it to our development board, a result of the device-specific nature of embedded programs rather than any limitation of our system. We selected nine firmwares from well-known open source repositories—Arduino Projects [1] and Raspberry Pi Pico Examples [9]—and the remaining four from the Fuzzware Open Source Data Set Fuzzware [85]. As for Fuzzware, we select firmware that already contains spatial memory errors, making them ideal for testing EMBEDWATCH’s capabilities as they contain *seven* real vulnerabilities, showcasing its efficacy in practical scenarios. The chosen firmware projects offer diversity, reflecting the simplicity often encountered in various embedded programs. When needed, we make modifications to the firmware source code to ensure library compatibility and support different architectures. More in detail for our experiments, we created two datasets: D_p for evaluating performance and false positive rate, and D_s for testing the system’s effectiveness. For D_p we carefully selected 11 embedded programs that exhibit the same behavior of polling sensors and performing specific actions on the data retrieved. This standardized approach allows us to create a reliable and consistent benchmark for our experiments. D_s , instead, consists of 15 programs, including the 11 programs from D_p , deliberately modified to introduce specific vulnerabilities, with one vulnerability introduced for each firmware. Additionally, the four remaining Fuzzware firmwares already contain genuine spatial memory vulnerabilities. More detailed information on

the types of vulnerability introduced in the firmware can be found in subsection 3.5.1.

3.5.1 RQ1: Bug Analysis & False Positives

Synthetic Vulnerabilities To rigorously assess EMBEDWATCH’s capabilities in detecting attacks and performing vulnerability analysis, we deliberately inject *eleven* various types of vulnerabilities into the system’s source code (*11* firmware), which include: (1) Control-Flow Hijacking: We overwrite stack variables to disrupt the firmware’s control-flow. (2) Global Variable Corruption: Intentional corruption of global variables within the code. (3) Struct Field Overwrite: The specific fields in `struct` data structures are overwritten. (4) Heap Variable Overwrite: Targeting and modifying variables residing in the heap. Our objective is twofold: to evaluate EMBEDWATCH’s resilience against both control-flow attacks, which manipulate process execution flow, and data-only attacks that exploit a program without altering its execution flow. To thoroughly examine vulnerability analysis, we strategically position vulnerabilities and attack points randomly in different code locations. The vulnerability data set is documented in [3]. To reach and trigger these vulnerabilities, we utilize a symbolic engine called angr [90], which generates the attack inputs for each firmware.

For each attack, EMBEDWATCH successfully detects the overflow attack and provides the exact point in the source code where the vulnerabilities were injected, along with the buffer involved in the operation. We manually inspect the execution trace to validate the crash report. Our experiment demonstrates the fundamental functioning of our prototype and confirms that EMBEDWATCH is a viable method for remote verifiers, offering precise vulnerability analysis. In addition, we conducted a thorough evaluation of false positives in our system by running our firmware data set with the input sets outlined in Table 3.1. The test inputs were generated using a symbolic executor using a code coverage metric. We closely monitor the generated execution traces of more than 1,300 inputs and observe no instances of false alarms.

Real-World Vulnerabilities Utilizing the Fuzzware dataset’s crash analysis section [6], we carefully selected four firmware samples: HEAT PRESS, PLC, THERMOSTAT, and RF DOOR LOCK. These chosen firmwares contain *seven* spatial memory errors, with HEAT PRESS and PLC exhibiting overflows on a field attribute, while the others suffer from stack buffer overflows. Although the Fuzzware data set provides input to trigger vulnerabilities, we enhanced the source code with instrumentation, leading to slightly different memory alignments compared to the original firmware. To address this, we used the QEMU system for debugging and identifying an optimal new input set that worked effectively with our instrumentation.

Deploying the firmware on our board, we validated EMBEDWATCH’s capability to accurately infer the root bug code location. EMBEDWATCH detected all spatial memory errors and automatically generated identical crash reports as those already reported in the Fuzzware dataset. This experiment significantly showcases EMBEDWATCH’s effectiveness in detecting real-world vulnerabilities across different classes, including dynamic memory allocation, global buffer, and local buffer vulnerabilities.

3.5.2 RQ2: System Usability & Performance

Unit Test To thoroughly assess the reliability and comprehensiveness of EMBEDWATCH, we conduct an extensive evaluation by generating a large corpus of inputs and their corresponding expected outputs. To ensure an unbiased input generation process, we employ symbolic execution, which allows us to automatically create safe inputs that uniformly cover various code sections within the firmware. Our objective is to maximize the coverage of the code in terms of the executed instructions for D_p . For this purpose, we rely on `angr` [90], which provides automatic code coverage input generation. Table 3.1 details statistics regarding the inputs generated.

Macrobenchmark As a crucial performance metric, we assess the influence of our solution on the run-time performance of the firmware. Specifically, we conducted three sets of measurements: one without EMBEDWATCH (baseline), another with EMBEDWATCH, and a third with EMBEDWATCH along with our caching mechanism. These measurements enable us to present the columns “Performance Overhead”, vividly demonstrating that our approach introduces an average overhead of approximately 1.44%, which diminishes to around 0.23% when the cache is applied. We set the cache size to 128 elements as a representative value, while conducting an ad hoc experiment on the cache performance in the following paragraphs. The results show that EMBEDWATCH incurs an overhead range between 0.07% and 36.04%, and an average of 1.4% without cache, whereas enabling the cache drops the overhead to a range between 0.01% and 2.33%. Further investigation of the sources of overhead reveals two different scenarios. In some firmwares, such as `CLOCK_PHONE`, `LEDMATRIXPAINTER`, and `HUE-MOTION`, the overhead is proportional to the number of instrumentation points executed at runtime. This relationship between overhead and the number of instrumented variables can be estimated by the results of our framework analysis (subsection 3.4.2). The more instrumentation points are present, the greater the overhead. However, there are firmwares that do not follow this trend (*e.g.*, `SYRINGE_PUMP`), which exhibits limited overhead despite a high number of instrumentation points. This is because these firmwares make use of idle operations, such as sleep or loops, that extend and skew the execution time. For example, `SYRINGE_PUMP` uses a sleep function for liquid

calibration. The use of idle operations in the firmware is a common practice, as many sensor data are read by waiting for events or physical components to complete. The usage of idle operations accounts for the discrepancy between the number of executed instrumented points and the relative overhead in some firmware. Finally, Table 3.1 shows the average trace validation time for each firmware based on the selected inputs, the trace validation is calculated with firmware without cache to show the worst-case scenario. We also implemented communication encryption to guarantee the confidentiality and integrity of the data exchanged between the firmware and the remote verifier. This approach protects against potential in-transit data manipulation and can detect unauthorized access by malicious attackers. The EMBEDWATCH protocol uses the AES CBC symmetric encryption algorithm with a 128-bit key size. The key is preloaded onto the device during the deployment phase and is also stored in the Verifier. As illustrated in Table 3.1, encryption introduces a small overhead of 0.4% in firmware execution time compared to the unencrypted counterpart. It is noticeable that firmware with larger TrustZone interaction and lower execution time has higher overhead in the encryption stage because it has more data transit to the verifier that needs to be encrypted.

Microbenchmark The goal of this evaluation is to identify the main source of overhead in EMBEDWATCH. This overhead is caused by the interaction between the firmware and the Trusted Application (TA) during run-time tracing. The firmware uses a trampoline library to call two secure functions, `lib_def_use` and `lib_flush_cache`. The first function, `lib_def_use`, tracks normal USE/DEF/IN interactions and transfers a small amount of data to the TA. The second function, `lib_flush_cache`, is used for cache optimization and performs bulk loads into the TA. We execute each secure function of the TA 1,024 times and compute the average overhead. Since the execution of `lib_flush_cache` varies according to the cache size, we repeat the experiments with 32, 64, 128, 256, 512, 1,024, 2,048, and 4,096 entries in the cache.

The overall execution time of the function `lib_def_use` was found to be 1,612 ms with a standard deviation of 0.60. In contrast, the overhead of the function `lib_flush_cache` depends on the cache size, which ranges from 1,548 to 3,298 ms. In the worst-case scenario, having 4,096 entries takes approximately twice as long as transferring a single entry with `lib_def_use`. The difference in performance is attributed to the context switch, which is the main source of overhead for EMBEDWATCH.

Firmware Size The firmware size increase results from the addition of instrumentation points and trampoline library calls during compilation. On average, a 40% increase in size is observed, contrasting with the lower over-

head exhibited by OAT (17%) in terms of instrumentation points. This discrepancy is attributed to the additional code implemented to minimize runtime overhead through instrumentation. We were unable to replicate the BLAST results due to the lack of source code. However, BLAST [103] reported a 64% increase in size (best case w/o function inlining) with embench-iot dataset [5]. In general, we consider 40% increase reasonable, thus positioning our approach midway between OAT, which enforces partial CF attestation, and BLAST, which achieves higher precision through whole-program CF attestation.

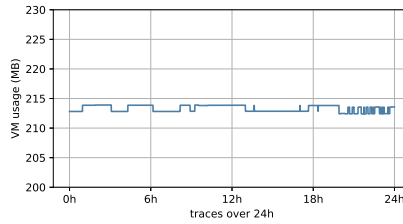


Figure 3.3: Virtual Memory (VM) allocated by the Verifier after 24h of interaction with a firmware.

Verifier Performance To assess the long-lasting stability of EMBED-WATCH, we carried out a comprehensive 24 hour experiment, running the Verifier continuously while a firmware continuously generated and sent traces. For this particular evaluation, we deliberately chose XML-PARSER as our test subject, given its participation in extensive memory operations, including stack and heap allocations, making it an optimal candidate for rigorous testing.

Throughout the experiment 3.3, the Verifier efficiently processed incoming traces without any noticeable delays. Regarding memory allocation, we observed that the Verifier quickly stabilized at a usage of 215 MB. Verifier memory consumption depends on factors such as the depth of the firmware call stack and the number of new memory chunks allocated, as discussed in section 3.4. This experiment demonstrates the Verifier’s stability during extended runs. Considering our intention to deploy the Verifier on high-performance machines, the allocation of 215 MB of memory is considered reasonable.

Performance Comparison with State-of-the-Art As EMBEDWATCH is designed for spatial memory errors detection leveraging remote attestation of ARM-based embedded devices, our primary points of comparison are OAT [93] and BLAST [103]. Although there are significant differences in the properties being attested — with EMBEDWATCH focusing on spatial memory violation detection and OAT and BLAST aiming to ensure control

Table 3.1: Number of dynamic instrumentation points and performance overhead. We measure the firmware execution time with cache enabled and with cache+encryption.

Firmware	Input	Dyn. Instr.			Performance overhead				Trace (byte) mean	Verify (sec)		Execution time			
		DEF	USE	Total	w/o cache (%)		w/ cache (%)			total	single entry	(msec) w/cache	w/encryption w/cache (msec)	encryption overhead (%)	
					mean	std. dev.	mean	std. dev.							
pixelPainter	43	6	13	19	0.64	0.0097	0.16	0.0079	254	0.05	8×10^{-5}	866.52	868.03		0.1742
clock_phone	10	1363	470	1833	36.04	42.2026	2.33	2.6471	292	0.05	5×10^{-5}	1300.4	1343.06		3.2810
ledmatrixPainter	100	46	100	146	10.06	0.0756	0.74	12.3331	2314	6.10	7×10^{-3}	603.77	604.41		0.1060
rc	10	50	51	101	0.23	0.1258	0.01	0.0025	5780	0.52	2×10^{-5}	10893.07	10895.39		0.0212
hello	3	26	2	28	1.26	0.3664	0.32	0.0057	448	0.13	7×10^{-5}	10893.07	10895.39		0.0212
mini-ig-stats	3	12	33	45	1.79	0.2223	0.22	0.0250	224	0.05	6×10^{-5}	722.77	723.79		0.1411
disco-keyboard	49	4	11	15	0.07	0.0058	0.00	0.0054	276	0.06	8×10^{-5}	1017.95	1019.43		0.1453
hello_uart	5	3	8	11	0.35	0.0088	0.12	0.0156	206	0.22	34×10^{-5}	1053.50	1056.00		0.2373
music-controller	1024	240	632	872	2.04	0.0031	0.05	0.0007	35313	3.60	3×10^{-5}	10104.49	10107.12		0.0260
syringe_pump	10	1474	895	2369	0.38	0.2113	0.00	0.0000	2427	0.30	4×10^{-5}	99.10	149.40		50.7568
hne-motion	46	64	88	152	19.31	8.0887	1.44	0.4508	143	0.05	9×10^{-5}	402.93	404.52		0.3938
Geometric mean	-	45.163	52.863	111.625	1.411	0.122	0.228	0.035	784.603	0.211	9.6×10^{-5}	711.46	751.46		0.4076

Table 3.2: Comparison of both static and dynamic calls to trampoline libs of OAT and EmbedWatch. For “Dynamic (TZ calls)”, we compute the arithmetic average of the trampoline invocations during the testing phase.

Firmware	OAT		EmbedWatch		
	Static	Dynamic (TZ calls)	Static	Dynamic (trampoline calls)	Dynamic (TZ calls)
heat_press	52	234.7	10	33	6.0
music-controller	36	1176.6	30	872	15.0
pixelPainter	22	160.0	33	19	6.0
rc	57	1743.0	10	101	6.0
syringePump	75	1730709.6	23	2370	106.8
clock_phone	37	120.0	48	1833	94.2

flow integrity — this performance comparison is still extremely valuable. We demonstrate that the overhead associated with our system is in line with that of other embedded system detectors.

Given that BLAST lacks open source code, our comparison was confined to OAT. We based our evaluation on the number of TrustZone invocations, both statically inserted during the compilation phase and dynamically triggered during the testing phase. These data serve as a suitable basis for comparison, since both frameworks exhibit a similar overhead in their interactions with the secure world. To collect these data, which are reported in Table 3.2, we modify the OAT trampoline libraries to count interactions with the Trusted Application (TA). This allows us to streamline the emulation process while retaining the data of interest. In addition, we add simple trace capabilities. Subsequently, we build the OAT’s compiler, the trampoline libraries, and test our dataset compiled with OAT tools. We exercise the firmware code with the same inputs as listed in Table 3.1 and record the number of instrumented points hit by the execution.

The results show that EMBEDWATCH exhibits a significantly lower number of calls compared to OAT. The disparity arises because OAT is heavily dependent on TrustZone calls to collect information on conditional branches within the firmware code, which is essential for accurate CFI attestation, as

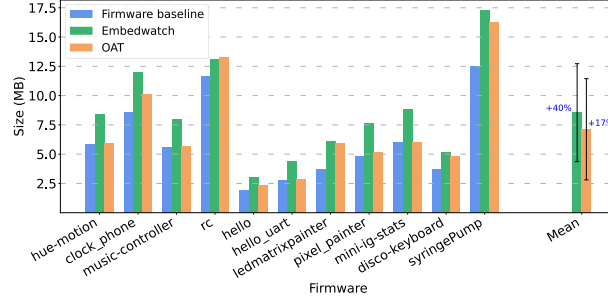


Figure 3.4: Comparison of instrumented binary size between EMBEDWATCH and OAT.

reported in [93]. In contrast, our system focuses solely on tracking the memory structures that depend on the input.

In our case, it is crucial to acknowledge that every time a call to the TrustZone is made, we pass more data to the TA compared to the OAT system. This arises from the presence of the cache, which effectively reduces the number of context switches with the TA, representing the primary source of overhead in such systems. The benefit of using the cache has already been studied in subsection 3.5.2. We report the instrumentation without cache (trampoline calls) and with cache (TZ calls) in Table 3.2. Similarly to subsection 3.5.2, we set the cache to 128 elements. We conclude that EMBEDWATCH delivers efficiency on par with the leading state-of-the-art solutions in the embedded systems landscape.

3.5.3 RQ3: Security Analysis

Our threat model takes into account the possibility that attackers discover and exploit previously unknown vulnerabilities in the embedded programs running in the Normal World. However, we assume that code injection or modification is not possible, which is already prevented by existing code integrity schemes for embedded devices [34, 65]. To bypass EMBEDWATCH, an attacker would have to take one of the following actions: (a) disable the instrumentation or trampolines, (b) misuse the instrumentation parameters, (c) exploit the interfaces that the measurement engine exposed to the trampolines, and (d) alter the execution trace, including replicating a previous attack. In the following, we examine each case separately.

(a) To disable the instrumentation, the attacker would need to modify the code, which is not possible because we assume that code integrity measures are in place. To evade the instrumentation, the attacker would need to take control of the control flow of the application, but since the instrumented code is executed before such actions, it is recorded in the execution trace.

(b) To successfully compromise the parameters passed to the trampoline functions, an attacker would need to overwrite the node IDs, the stack index, the buffer base address or the buffer size (for HDEF). Our instrumentation is designed to protect against attacks against these components (subsection 3.4.3). We store the values of statically inferred information, such as node IDs, as immediates in the read-only section `.text` of the program during compilation. The values of runtime attributes, such as the base address and size of a buffer, are recorded in the corresponding DEF events immediately after the buffer is created. Therefore, we store the base address and size in the TA before an overflow or attack occurs. For IDs that refer to buffers passed as function pointers and the stack index, we adopt a solution similar to SafeStack [10]: a randomly allocated area that contains sensitive variables (*i.e.*, buffer ID and stack index). We also employ a similar solution to SafeStack for protecting loop optimization. In this case, we maintain a cache that is randomly allocated in the Normal World, which is flushed in the TA immediately after the loop ends. Since we assume that the loop does not contain indirect jumps or function calls, a payload cannot attack the buffer before the loop terminates, while an overflow would not easily reach the cache.

(c) Implementing our compiler comes with a unique restriction: it disallows the use of SMC (Secure Monitor Call) instructions outside the trampoline library. This restriction is put in place to ensure the security and integrity of the system. It allows only the trampoline functions to invoke the Trusted Application (TA) interfaces, effectively disabling the Normal World from accessing them. By implementing this restriction, we can maintain the secure environment for the system and prevent unauthorized access to the TA interfaces.

(d) Our system ensures the integrity of the attestation blob through a signature generated by TEE using a hardware-provisioned private key. A verifier can easily check the signature and verify the integrity of the attestation blob. Replay attacks are also prevented by checking if the cryptographic nonce inside the attestation blob matches the one originally generated by the verifier.

3.6 Discussion

In this section, we discuss limitations of EMBEDWATCH and propose new challenges for future work.

Aliasing of local variables The EMBEDWATCH prototype tracks buffer IDs for variables passed as function arguments, including their aliases during runtime. However, it does not handle cases where a pointer stored in a static or global variable is subsequently aliased to multiple local variables, or where

the pointee of such a pointer is determined dynamically. To address this, we need to extend buffer ID propagation to local variables, detecting aliases at compile time, similar to function pointer parameters. In our experiments, we have not encountered any such cases, so the implementation of this feature is deferred to future improvements. However, it remains an important area of focus for further development.

Interrupts handling The current prototype does not integrate the interrupt handler trace. We can easily extend EMBEDWATCH to handle signals by using our graph model to analyze signal handlers containing attacker-controllable memory structures. Then the existing EMBEDWATCH design already identifies and reports OOB accesses within the signal handlers. Therefore, we consider the extension for interrupt handlers as a pure engineering task. Moreover, the current target firmware does not contain interrupt handlers with attacker-controllable input.

Bypassing Instrumentation Because EMBEDWATCH relies on an LLVM pass to instrument the firmware code before attestation, a key limitation lies in operations that bypass compiler-level instrumentation. The most notable cases are direct memory accesses performed by DMA engines and interactions with pre-compiled third-party libraries. In the first case, DMA transfers occur without executing instrumented instructions, meaning that writes performed by the peripheral are invisible to the verifier. A possible mitigation is to analyse the DMA configuration at the moment it is enabled and detect whether the programmed transfer could write outside the intended target region. For third-party libraries, the lack of available source prevents injecting instrumentation into their internal memory operations; however, function-call boundaries can still be modelled. By treating calls into such libraries as abstract memory operations, similar to how EMBEDWATCH handles functions like `scanf` or `gets`, it becomes possible to detect out-of-bounds effects that originate within the library and target application-level data. A full exploration and implementation of these mitigation strategies is left to future work.

Temporal memory safety EMBEDWATCH is primarily designed to identify and address exploitable spatial memory safety violations. However, we propose that its methodology and framework could be adapted to recognize exploitable temporal memory safety violations. This adaptation, while promising, introduces considerable challenges. A key obstacle is the need to monitor the liveness of dynamic memory allocations, a vital component in detecting temporal errors. Such monitoring, as previously discussed, has the potential to adversely affect system performance. Additionally, the development of an advanced pruning system is essential. This system should

mirror the functionality of the one in EMBEDWATCH, selectively instrumenting operations that pose a risk of leading to a security breach. This targeted approach is crucial to ensure the feasibility of the enhanced system in practical, real-world applications. The full implementation of this extension requires further research and development, representing a significant yet worthwhile direction for future advancements.

3.7 Conclusion

In this chapter, we introduce EMBEDWATCH, a novel crash reporting system designed for embedded devices. By combining fat pointer mechanisms with remote attestation, it addresses the distinct challenges of attack analysis in resource-constrained and real-time environments. Our system excels in detecting and analyzing memory errors, such as data-only and control-flow attacks, across different memory segments. Our experimental evaluations of real-world firmware and CWEs show that EMBEDWATCH is very accurate in identifying the root causes of spatial memory errors, with a minimal overhead range between 0.01% and 2.33%. These findings underscore the system’s effectiveness and suitability for real-world IoT environments.

Chapter 4

Optimizing Embedded Device Security with Scalable In-Place Fuzzing

4.1 Problem Statement

In the current digital era, computing extends beyond the traditional framework of desktops for daily activities and servers for demanding tasks. Instead, embedded devices are increasingly utilized in multiple industries, such as transportation, industrial automation, healthcare, and consumer electronics. Given their inherent resource limitations and the fact that they operate within safety-critical hard real-time environments, optimizing time and spatial efficiency is essential; however, ensuring security is of utmost importance. The increasing number of cyberattacks on embedded systems highlights the security challenges associated with the Internet of Things (IoT) and emphasizes the need for robust and systematic testing strategies. This deficiency does not stem from a lack of interest from the industry or academia but rather from the complex nature of embedded systems. Their architecture is vastly different from standard computing systems, making it difficult to apply general-purpose analysis tools effectively. The main issues include stringent resource limitations that hinder thorough analysis and the absence of typical features associated with general-purpose computers, such as filesystems, processes, or an operating system. Despite these limitations, the growing prevalence of embedded devices in safety-critical and resource-constrained environments has intensified the need for automated vulnerability testing techniques. However, testing methodologies developed for traditional computing platforms often fail to translate effectively to embedded environments due to their unique architectural constraints.

In response to these challenges, recent years have seen various efforts to adapt fuzzing techniques, one of the most effective automated vulnerability

detection methods, to embedded systems. These adaptations fall primarily into two broad categories: (1) *Emulation-based techniques*, which attempt to execute firmware in a virtualized or rehosted environment, allowing a more scalable and introspective fuzzing process. (2) *In-place fuzzing*, which executes test cases directly on the embedded device, ensuring interaction with actual hardware peripherals and reducing false positives and negatives. Each approach presents unique trade-offs. Emulation-based techniques (e.g., rehosting and firmware transplantation) allow researchers to efficiently analyze firmware across multiple devices without requiring physical hardware. These methods enable fine-grained introspection, aiding in the detection of subtle software vulnerabilities. However, their effectiveness is highly dependent on the accuracy of peripheral modeling, a fundamental challenge, since embedded devices often include proprietary, undocumented, or highly customized hardware components. Inaccurate peripheral modeling can lead to incomplete or misleading security assessments.

In contrast, in-place fuzzing provides a more realistic security evaluation by running test cases on the actual embedded device, interacting with real peripherals, and uncovering hardware-related vulnerabilities that emulation might miss. This approach minimizes false positives and negatives and ensures a more accurate assessment of the behavior of the system under attack conditions. However, it introduces significant challenges, particularly in terms of scalability and efficiency. Due to severe resource constraints, the in-place fuzzing must operate within strict memory, processing, and power limitations. Additionally, communication overhead between the test system and the embedded device can dramatically impact performance, requiring careful optimization to ensure a viable fuzzing rate. Instances of these systems include IPEA [89] and μ AFL, which exemplify hardware-in-the-loop methodologies operating directly on peripherals. IPEAFuzz employs minimalistic instrumentation to streamline data communication between the hardware and the fuzzer. In contrast, AFL exploits ARM-specific debugging capabilities for device interaction, eliminating the need for firmware instrumentation.

Our research seeks to enhance the scalability and performance of these solutions by sustaining AFL++ technology while remaining orthogonal to the current state-of-the-art approaches. The choice of focusing on improving the execution speed of in-place fuzzing stems from the fact that, for fuzzing as a whole, incrementing the number of tests per second directly amplifies the methodology’s bug-finding capacity, since the probability of triggering rare state transitions grows with the number of executed test cases. While emulation and re-hosting approaches facilitate introspection, they often fail to reproduce peripheral interactions and timing behavior, which are essential in firmware validation. This means that improving execution speed in in-place fuzzing can be necessary to efficiently discover classes of bugs tied to how peripherals handle input which can be harder to detect via re-hosting

or emulation based approaches. Nonetheless the proposed architecture complements, rather than replaces, such approaches by providing high-fidelity testing where accurate hardware behavior is required. To achieve this objective we introduce *E-FuzzEdge*, an in-place fuzzer designed to enhance execution efficiency and provide a scalable resource-sensitive testing framework tailored to embedded environments. Our project focuses on two main goals: first, to investigate methods for enhancing in-place fuzzing, thereby increasing the execution rate on the test device; second, to provide an open-source fuzzer to the community, which leverages cutting-edge technologies by adapting AFL++ for the IoT environment. In pursuit of these goals, we first developed a new fuzzing framework aimed at optimizing the use of embedded devices. This is achieved by minimizing the data transfer between the device that runs the inputs and the desktop responsible for handling feedback and producing new inputs. Our empirical observations identified this data exchange as the main source of overhead in hardware-in-the-loop fuzzing methods. Furthermore, we designed a parallel system that enables uninterrupted execution while desktop instances produce new inputs, aiming to increase the execution frequency by cutting down on the downtime typically encountered when the fuzzer gathers and analyzes the feedback data after the execution of the test case. We then created a prototype of this system that uses AFL++, which is freely accessible as mentioned in [47], refining the current remote fuzzing technology available while maintaining its original architectural functions as much as possible. Our prototype was assessed in comparison to two leading-edge in-place embedded fuzzers: IPEA [89] and μ AFL [31]. We noted a notable enhancement in execution speed, which illustrates that the proposed architecture has the potential to considerably boost the throughput of a fuzzing campaign. Moreover, the contributions of our model are orthogonal to both IPEA and μ AFL meaning that a similar architecture can be integrated into these projects to further improve execution speed while retaining their key contributions to the topic of embedded fuzzing.

4.2 E-FuzzEdge Design

The main focus of E-FuzzEdge is designing an efficient greybox in-place fuzzing architecture for embedded devices, minimizing execution overhead while maintaining adaptability across different hardware environments.

The core requirements of our architecture are versatility and efficiency. We can examine the former along two dimensions: the adaptability of the communication channel to suit the diverse environments typical of embedded devices, and a decrease in platform requirements, such as processes and filesystem. This adjustment is necessary for platforms with extremely simplistic or non-existent operating systems. In terms of efficiency, the main

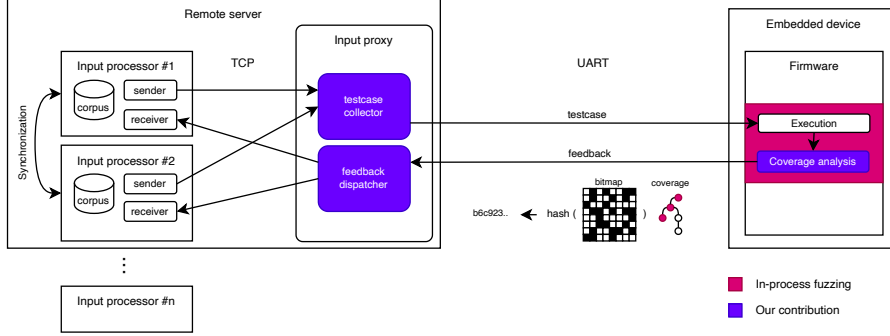


Figure 4.1: E-FuzzEdge Architecture

issue is the additional burden caused by the communication channel. As this burden is intrinsic to the platform, our method emphasizes reducing its impact instead of trying to remove it altogether. We address this overhead by employing two main strategies: decreasing the volume of data transmitted through the communication channel and reducing device idle time through enhanced data transmission and processing.

Increasing the execution speed of in-place fuzzing directly amplifies its bug-finding capacity, since the probability of triggering rare state transitions grows with the number of executed test cases. While emulation and re-hosting facilitate introspection, they often fail to reproduce peripheral interactions and timing behavior, which are essential in firmware validation. The proposed architecture complements, rather than replaces, such approaches by providing high-fidelity testing where accurate hardware behavior is required.

4.2.1 Architectural Model

We devised our architecture in order to satisfy the mentioned requirements, particularly the one related to overhead, since versatility is related to the implementation and not to the model. Our system remains consistent with standard forserver-based fuzzing designs described in Chapter 2, but we decouple input processing, mutation, and scheduling, from test case execution and feedback collection. Figure 4.1 shows the three elements our system is built upon: multiple *input processors*, a *input proxy*, and a single *input executor*

Input Processor Multiple processors on a desktop device, leveraging superior computing resources and filesystem access to enhance usability and accelerate operations that do not require direct interaction with the embedded hardware. They are tasked with all operations related to the processing of test cases, such as scheduling and mutation, based on coverage feedback.

The choice of using multiple *input processors* stems from the idea of maximizing the use of the device. As we said, sending and retrieving data on the communication channel is a slow process, the use of a one-to-many architecture ensures that the single executor does not remain idle while a processor is retrieving and analyzing the data, but runs the inputs provided by the other executors thus maximizing the throughput.

Input proxy A lightweight middleware sitting between processors and device. It exposes a stable host-facing interface and adapts to the device link (TCP or UART). For UART it handles practical details like DMA cyclic reception and padding so half/full-buffer interrupts fire predictably. The choice of a proxy comes with the benefit of easing the process of changing the communication protocol between the desktop system and the embedded device, which is particularly relevant in a highly heterogeneous context in terms of communication protocols.

Input Executor The executor runs in-process with the target firmware and a small harness without relying on processes or forking systems that can be absent when fuzzing bare-metal. The harness is tasked with initializing the necessary peripherals for testing the firmware, retrieving inputs from the proxy and feeding them to the program when needed. The execution of test cases follows the standard fuzzing loop of execution and feedback analysis but is a persistent process, meaning that the system is not rebooted after each test case. While this approach can create issues due to separate tests concurring in the stimulation of a bug, we argue that this issue can be mitigated by careful harness design and the benefit in terms of executions per second makes adopting the persistent approach worth it. The executor is also tasked with processing coverage data related to the executed test, this allows us to minimize the data transmitted on the communication channel, in the form of a flag indicating whether new bits were found and a checksum, and avoid performing full-bitmap transmission that would significantly hinder fuzzing performance.

4.2.2 Communication Protocol

The protocol mirrors the classic fuzzing loop of schedule, mutate, execute, evaluate, but is engineered to minimize bytes in flight and device idle time. During initialization, processors establish TCP sessions with the proxy, the proxy opens or attaches to the device link, and the executor configures its peripherals and allocates the coverage structures it will maintain during the campaign. Any static configuration that affects message formats, such as the effective map size or maximum payload, is broadcast from the proxy to processors so formatting remains consistent.

In steady state each processor prepares a test case and submits it to the proxy; the proxy forwards the frame to the device; the executor runs the harnessed target to completion or timeout, computes coverage locally, and performs on-device triage against the cumulative map. It then sends a compact feedback record composed of a fault code when applicable, a boolean flag indicating whether new bits were observed, and, if enabled, a short checksum. The proxy routes this record back to the originating processor, which decides whether to retain or discard the input and immediately schedules the next one. Because only minimal feedback travels on the hot path, steady-state traffic remains small and predictable.

Transport specifics are handled transparently by the proxy. Over UART, the executor configures DMA in cyclic mode and relies on half and full-buffer interrupts to delimit frames, while the proxy pads headers and bodies so those interrupts fire deterministically.

4.3 Implementation

In order to test our model, we developed a prototype based on AFL++ [47]. Although AFL++ already provides a proxy system, we developed the remote fuzzing system from scratch because that system allows only a one-to-one fuzzing architecture and requires the fuzzer to send back the whole coverage map, significantly slowing down the whole process. Additionally, we modified the forkserver to allow remote communication with the embedded device and adapted the feedback collection mechanism to process the input without the need for the coverage map, since the task of analyzing the input is performed by the input executor. Moreover, we modified the compiler runtime to accomplish two objectives: first, we reduced the data needed for the fuzzing campaign in order to correspond to the strict space requirements present within the embedded context; second, we added the infrastructure necessary to initialize the peripherals needed for communicating with the proxy and retrieve input data from it. Lastly, we developed the proxy system to mediate between the afl instances and the embedded system. In total, we contributed 6534 lines of C/C++ code.

4.4 Evaluation

We evaluate E-FuzzEdge in two settings: a desktop-only setup where input processors and the executor communicate over TCP and a hardware-in-the-loop setup using an STM32L053R8 board as the embedded device and a desktop host (Intel i7-9750H, 16 GB RAM) performing seed processing. In the embedded setup, host-device communication uses UART.

For each firmware in our dataset, we ran an average of seven fuzzing campaigns of 4 hours each for $n \in \{1, 2, 3, 4\}$ parallel input processors. We

report executions per second (exec/s) aggregated over each campaign. The 4-hour duration allows the fuzzer to stabilize while keeping runs practical.

4.4.1 Dataset

On desktop we used `xpdf`, a standard fuzzing target. For the embedded evaluation we reused the dataset from Chapter 3. Each embedded firmware was slightly modified to remove the original `main` and to install a simple, in-process harness (standard practice in fuzzing). Common input functions (e.g., `fgetc`, `scanf`) were reimplemented to read from the proxy-provided test case rather than `stdin` or UART, leaving application logic unchanged.

The initial corpus for each fuzzing campaign consisted, when available, of the test samples distributed with the target binary, for example, the sample files provided with `Xpdf` or the XML snippets bundled with the XML parser. When no such samples were available, we manually crafted a minimal but representative input that exercised the expected functionality of the appliance. We deliberately avoided using empty inputs, as they do not reflect realistic pre-deployment testing conditions and typically fail to reach meaningful paths, both in embedded firmware and in general-purpose fuzzing workloads.

4.4.2 Results

Desktop Setting To bound the potential benefits of our host-side concurrency model under a fast link and a high-performance executor, we replicated the client-proxy protocol over TCP on a desktop host and executed the target with a single forkserver-backed *input executor*. With one executor, baseline throughput is **20.24 exec/s**. Allowing multiple executors in parallel yields speedups of $2.09\times$, $2.10\times$, $3.20\times$, $5.11\times$, and $6.81\times$ for $\{2, 3, 4, 6, 8\}$ executors, corresponding to absolute throughputs of $\{42.28, 42.44, 64.69, 103.45, 137.96\}$ exec/s. It is clear that as concurrency increases, scaling trends toward CPU-bound behavior: improvements persist until available cores and memory bandwidth saturate, at which point speedup becomes sub-linear.

This configuration is a stress test rather than a recommended desktop setup: general-purpose binaries can already fork freely without our proxy, and running multiple independent fuzzers per core is standard practice; on typical desktop targets, execution rates are *on the order of thousands of exec/s*, i.e., at least an order of magnitude higher than our proxy-mediated desktop measurements even after adding parallelism. Nevertheless, the experiment is informative for embedded scenarios because it isolates the effect of *low-latency transport plus host-side concurrency* from device constraints. In other words, it approximates the leeway available to high-performance boards connected over fast links where the communication channel does not

dominate. Two practical takeaways follow. First, when the link is fast enough, parallel producers convert directly into higher executions per second until the host CPU, not the channel, becomes the bottleneck. Second, the exact “knee” depends on the balance between per-input compute in the target and execution costs. This behavior mirrors our embedded results, where the knee appears earlier because the on-device executor and slower link saturate sooner. Overall, the desktop scaling curve provides an optimistic upper bound on the gains a high-performance board and transport could realize under the same one-executor/many-producer architecture.

Embedded Setting Table 4.1 reports executions per second (exec/s) by firmware and by the number of parallel processors feeding the device. Values in parentheses are per-target speedups relative to one processor, and the “Total” row is the geometric-mean speedup across firmwares for that column (targets without data are excluded).

The headline result is that adding a *second* processor consistently improves throughput: the geometric-mean speedup across firmwares is **1.27×**. A third processor yields 1.06× on average, and a fourth 0.95×, indicating diminishing—and sometimes negative—returns beyond two. Concretely, 7/10 firmwares improve at two processors (typical gains ≈ 1.3 – $2.0\times$), 2/10 are unchanged, and 1/10 regresses due to internal delays. This pattern reflects the intended division of labor: a second producer keeps the executor busy while the first is scheduling and processing feedback, but additional producers mostly contribute queueing and per-message overhead once the device and link are saturated.

Saturation is target-specific. The best-performing n is 2 for `discokeyboard`, `huemotion`, `rflock`, and `xml` (4/10); 3 for `ledmatrixpainter`, `thermostat`, and (effectively) `musiccontroller` (3/10); 4 for `modbus` (1/10); and 1 for `miniigstats` and `pixelpainter` (2/10), where fixed delays cap exec/s regardless of host-side concurrency. These outcomes align with intuition about where time is spent: when per-input device time and channel overhead dominate, extra processors add little beyond contention; when the target spends more time per input (e.g., protocol parsing in `modbus` or deeper control-flow in `thermostat`), a second processor converts directly into near-linear gains, with limited benefit thereafter. Operationally, on UART-class links and STM32L0-class boards, two processors are a strong default; the precise knee shifts with firmware behavior and transport latency/bandwidth, but the guiding principle remains to minimize bytes on the wire, keep the executor in a persistent loop, and scale concurrency on the host until the device or link becomes the bottleneck.

Table 4.1: Number of executions per second for each firmware

Firmware name	1 Instance	2 Instances	3 Instances	4 Instances
discokeyboard	14.36 (1.00x)	21.17 (1.47x)	10.03 (0.70x)	10.42 (0.73x)
huemotion	14.31 (1.00x)	21.68 (1.52x)	10.03 (0.70x)	10.08 (0.70x)
ledmatrixpainter	18.59 (1.00x)	23.71 (1.28x)	23.93 (1.29x)	12.01 (0.65x)
miniigstats	7.41 (1.00x)	4.64 (0.63x)	4.67 (0.63x)	4.60 (0.62x)
modbus	16.79 (1.00x)	25.76 (1.53x)	26.07 (1.55x)	27.89 (1.66x)
musiccontroller	0.94 (1.00x)	0.94 (1.00x)	0.95 (1.01x)	0.94 (1.00x)
pixelpainter	2.62 (1.00x)	2.62 (1.00x)	2.61 (1.00x)	2.61 (1.00x)
rflock	15.89 (1.00x)	23.94 (1.51x)	21.94 (1.38x)	20.00 (1.26x)
thermostat	16.62 (1.00x)	32.96 (1.98x)	33.32 (2.00x)	24.70 (1.49x)
xml	3.22 (1.00x)	4.17 (1.30x)	-	-
Total	1.00x	1.27x	1.06x	0.95x

4.5 Discussion

Our evaluation has demonstrated the effectiveness of our parallel architecture, but it also highlighted several key limitations and avenues for future work.

Evaluation scope. As shown in the previous section, E-FuzzEdge improves fuzzing throughput when multiple input processors handle test cases in parallel. Our evaluation, however, is limited to a single MCU family (STM32L0) and two channels (TCP on desktop, UART on device). While we do not expect an inversion of the observed trend, more powerful boards and faster links will likely shift the saturation point beyond two processors. Replicating on additional MCUs and transports (e.g., USB CDC/SPI) is future work.

Memory overhead. Our prototype maintains one temporary coverage map per execution and one cumulative map per input-processor instance on the device. On memory-constrained boards with large targets, these allocations may be prohibitive. A straightforward extension is to use a single cumulative map shared across processors, reducing redundancy; this needs careful evaluation to rule out contention and other side effects.

Number of input processors. The current parallelism level is fixed at campaign start. Results indicate the optimal number of processors depends on both firmware and target hardware. On STM32L053R8, two processors are typically best. We plan to improve on this area by enabling runtime negotiation of the number of input processors, this approach can increase/reduce their number dynamically to keep the board saturated while minimizing the overhead caused by the communication channel in order to optimize the system throughput.

Crash Handling Currently, we rely on timeouts to detect crashes. While this is a common and practical approach, it cannot distinguish a true crash from a simple timeout due to a delay in the target firmware’s execution. This limitation can be addressed by implementing custom error handlers within the firmware to signal the host when a crash occurs. Furthermore, our architecture’s high throughput, while beneficial, also exacerbates a core problem of in-place fuzzing: the need for frequent device resets upon crash detection. An increased crash discovery rate, while a sign of a successful fuzzing campaign, can lead to more frequent reboots, which in turn can significantly impact overall performance and limit the practical benefits of our architecture.

Hard-to-Reach Firmware Locations Finally, increasing execution throughput only improves bug-discovery potential in a probabilistic sense. While higher test-case generation rates help uncover rare events or peripheral-driven behaviours, they do not fundamentally overcome cases where the vulnerable code region is guarded by highly complex path or semantic constraints. In these scenarios, the dominant barrier is the fuzzer’s ability to satisfy a specific sequence of control-flow decisions or state preconditions, rather than the number of executions per second. As a result, the benefits of E-FuzzEdge can be diminished when reachability is structurally limited, an inherent challenge of coverage-guided fuzzing rather than a limitation specific to our system.

4.6 Conclusion

In this work, we introduced E-FuzzEdge, an in-place fuzzer for embedded systems that leverages a novel parallel architecture to maximize fuzzing throughput. Our primary finding is that by decoupling input processing from on-device execution and feeding a single executor with multiple parallel input processors, we can significantly boost the execution rate. To demonstrate this, we developed an open-source prototype based on AFL++ that successfully implements our proposed architecture within a state-of-the-art greybox fuzzer. We evaluated our prototype with a set of different firmwares and observed a notable geometric mean speedup of 1.27x with just two parallel processors.

This result is particularly significant because our approach is orthogonal to the core contributions of existing in-place fuzzers like IPEA and μ AFL. Our architectural model can be seamlessly integrated into these and other similar frameworks, enabling them to achieve higher throughput and, consequently, more efficient vulnerability discovery.

By bridging this gap, we hope to provide the community with both a valuable tool for enhancing the security of resource-constrained devices and

a novel architecture capable of boosting the performance of future on-device embedded fuzzing prototypes in general.

Chapter 5

Bug Complexity and their presence in fuzzing benchmarks

5.1 Problem Statement

The previous two chapters addressed *how* to run more effective experiments on embedded software: at runtime, EMBEDWATCH showed that commodity devices can report precise manifestation sites of spatial memory errors with negligible overhead, and, pre-deployment, E-FUZZEDGE demonstrated that host-side concurrency with a single on-device executor can substantially raise executions per second under realistic links. While these results strengthen the mechanics of *where* failures happen and *how fast* we can explore, they also expose a deeper question that throughput alone cannot answer: *what makes some bugs hard to trigger in the first place?*

Both on general-purpose and embedded systems fuzzing is far from a solved problem. Critically, many faults only occur when a specific combination of control-flow and data-flow conditions is met—requiring a fuzzer to not only “reach” a vulnerable location in code, but also to follow the exact branching decisions and meet subtle semantic prerequisites that trigger the bug. In practice, this means that some vulnerabilities remain stubbornly hidden behind layers of logic, escaping even state-of-the-art fuzzers for months or years.

Understanding exactly *why* some bugs evade fuzzing—and quantifying the obstacles that a fuzzer must overcome is therefore crucial if we hope to improve the effectiveness of these tools. For instance, we can consider two examples. In the first, the fuzzer never reaches the vulnerable instruction because a series of nested `if`-conditions prevent unstructured input from ever arriving at that target code region. No matter how many mutated test cases the fuzzer generates, the fuzzer is never able to “hit” the bug’s

location. In the second example, the fuzzer reaches the correct code area, but it may traverse an incorrect branch or fail to satisfy an implicit state predicate, such as a particular data-structure layout, before the fault occurs. In this case, while the vulnerable code is tested, the right combination of control-flow and data-flow constraints needed to activate the flaw remains out of reach.

Existing fuzzing benchmarks and prior studies often treat all unfound bugs as simply “hard,” without dissecting the underlying reasons for their elusiveness [57, 40, 24]. For example, MAGMA [57] and many OSS-Fuzz evaluations [40] simply count whether a bug was found within a time limit, labeling any timeout as a “hard” bug. Similarly, in their standard benchmark comparisons, researchers assume that the absence of a crash implies a single category of difficulty, rather than distinguishing between unreachable code or unmet state constraints [24]. This lack of clarity makes it difficult to evaluate new fuzzing techniques or to build more representative test suites; for instance, a benchmark suite might include many bugs that are unreachable under typical fuzzing strategies, but few that hinge on subtle data-state conditions. Since these datasets are then used to compare fuzzers and drive research in the area, a bias towards a certain class of complexity might steer the entire field to overfit on them and thus leaving other types of bugs undetected.

To address these gaps, in this chapter we introduce a *rigorous framework for quantifying bug complexity* from the perspective of coverage-guided, grey-box fuzzing. By decomposing the bug-triggering condition into three orthogonal dimensions—(1) *reachability complexity*, (2) *path complexity*, and (3) *state complexity*—we can precisely measure which constraints impede fuzzers in practice and their prevalence in popular vulnerability datasets. Reachability complexity captures the difficulty of simply executing the vulnerable instruction at least once. Path complexity encompasses the challenge of selecting the correct branches so that once the fuzzer reaches the target location, it also satisfies the conditional logic required for the fault to occur. State complexity reflects any additional semantic prerequisites—such as data-structure invariants and numeric relationships—that must hold at the vulnerable site beyond those already expressed in control-flow statements.

Our approach to measure the complexity of these components unfolds in three stages of targeted fuzzing campaigns. First, we measure how long it takes a fuzzer to reach each vulnerability, ignoring deeper constraints. Next, we enforce that any input reaching the vulnerability must also follow the correct branch sequence—thereby isolating *path complexity*. Finally, we constrain the fuzzer to inputs that already handle control-flow correctly and observe how long it takes to satisfy any remaining *state complexity* conditions. By comparing these campaigns across a large set of real-world bugs—from the MAGMA collection and curated OSS-Fuzz examples—we

are able to identify which dimension most often constitutes the primary barrier.

Our experiments confirm three overarching findings about real-world bugs in MAGMA and OSS-Fuzz:

- **Reachability is often straightforward.** In MAGMA, more than 75% of bugs were reached within just a few hours when using the default seed corpus, and all “fast-discovery” OSS-Fuzz bugs showed trivial reachability. Only about 25% of MAGMA bugs—and roughly 40% of the slower OSS-Fuzz bugs—required substantial effort just to execute the vulnerable instruction.
- **Path complexity is the dominant barrier.** Nearly half of MAGMA bugs were never triggered, even after five days, when starting from inputs known to reach the vulnerable location. Likewise, 80% of the “hard” OSS-Fuzz bugs exhibited very high path complexity. In other words, once reachability is guaranteed, most bugs still evade detection because the fuzzer cannot follow the exact series of branch conditions needed to activate the fault.
- **Hidden state constraints are rare.** Only about 17% of MAGMA bugs and 35% of OSS-Fuzz bugs required any non-trivial state condition beyond control-flow. Even then, those state predicates were typically satisfied quickly (within a few hours). This results are concerning if compared with state-of-the-art works that, by adding state-sensitivity to their fuzzer, discovered new bugs in already fuzzed code bases [46, 22].
- **Edge dependencies correlate with success.** Over 80% of bugs that were successfully retriggered contained at least one “discriminant edge” (a branch common to all crashing paths that does not appear in non-crashing ones). By contrast, fewer than 50% of non-retriggered bugs had such an edge, suggesting that edge-sensitive coverage feedback significantly helps the fuzzer to discover the correct path in the vulnerabilities we tested. However, as we discuss in the paper, this correlation might also just be a consequence of the fact that we use vulnerabilities discovered by fuzzers (often relying on edge-coverage as feedback) to create the datasets in the first place.

Together, these results show that, although reaching a vulnerable location is sometimes nontrivial, the primary obstacle is almost always navigating the precise control-flow path. While state-level constraints appear secondary in our measurements, this likely reflects a *survivorship bias*: widely used datasets and the engines that discover them co-evolve toward bugs that are path-dominant and comparatively state-light, underrepresenting

genuinely stateful vulnerabilities. The implication for *fuzzing at large* is twofold. First, evaluation should move beyond undifferentiated “hardness” to report, per bug, the relative contributions of reachability, path, and state, and benchmark suites should be curated to include faults whose triggering predicates hinge on non-trivial program state (protocol phases, multi-step invariants), so that state-aware guidance is meaningfully exercised rather than implicitly discounted. Second, design effort should prioritize richer *path-aware* feedback (e.g., context/history sensitivity, path-set diversity) while also developing lightweight mechanisms to represent or learn useful state when it does matter. For *embedded fuzzing* in particular, where compute budgets and links are constrained, the taxonomy helps allocate effort where it pays off. In this way, the present framework complements the preceding chapters: once throughput and observability have been improved, it explains *why* residual failures persist and clarifies *where* to invest next.

The remainder of the Chapter is organized as follows. Section 5.2 formalizes our three-dimensional taxonomy of bug hardness and describes the theoretical underpinnings of deconstructing the bug-triggering predicate. Section 5.3 details our methodology for isolating each complexity dimension through targeted fuzzing campaigns. In Section 5.4, we explain our implementation of enhanced instrumentation and crash deduplication in AFL++. Section 5.5 presents the results of our experiments on the MAGMA and OSS-Fuzz benchmarks, including an analysis of edge dependencies. Section 5.6 and Section 5.7 discuss the broader implications of our findings and potential limitations and threats to validity. Finally, Section 5.8 concludes with a summary of contributions and a look ahead.

5.2 Taxonomy of bug complexity

In this section, we define the notion of *bug* and *bug complexity* as it pertains to dynamic fuzzing, and present a formal framework to reason about it. Our key insight is that the difficulty of triggering a vulnerability can be decomposed into three distinct but interrelated components: *reachability*, *path*, and *state* complexity. This decomposition enables structured analysis and empirical measurement of fuzzing difficulty at a per-bug level.

5.2.1 Defining Bugs

Throughout this work, we will discuss the complexity of bugs in the context of dynamic fuzzing. It is important to clarify that our definition of a bug differs from the typical software engineering definition.

In conventional development, a bug is a specific coding mistake or logical flaw that exists independently of whether it is ever triggered. By contrast, for a fuzzer a bug is inseparable from its *manifestation* during execution.

The fuzzer’s goal is not to reason about root causes in the source code, but to find inputs that drive the program into an *observable failure state* (e.g., crash or hang). Consequently, when we refer to the “bug location,” we mean the code site where the failure *manifests* (e.g., the faulting PC), which may differ from the underlying origin of the defect. Following IEEE definitions [14, 13] we identify the bug, and its location, as the line of code where a fault takes place and not where an anomaly originates.

This distinction matters for two reasons. First, in grey-box fuzzing the fuzzer cannot “see” latent defects, only their effects under execution oracles. Second, the relevant challenge is not whether a defect exists, but how hard it is to *trigger* it. Our taxonomy therefore measures the difficulty of producing a witnessing input by decomposing it into reachability, path, and state components.

5.2.2 Defining Bug Complexity

We broadly define *bug complexity* as the difficulty, from the perspective of a dynamic testing tool, of triggering a known vulnerability within a program. This notion is not absolute — rather, it is tightly coupled to the capabilities and limitations of the testing approach in use. A bug that is easily triggered via symbolic execution may remain undetected by a black-box fuzzer. Conversely, bugs that require minimal control or data constraints might be discovered trivially by even the most naive mutational techniques.

In this work, we focus specifically on dynamic fuzzing — particularly grey-box, feedback-guided fuzzers — given their widespread adoption in software security testing. Our goal is to characterize why some bugs are significantly more difficult to reach and trigger than others, and to identify the sources of such complexity in a principled way.

5.2.3 A Formal Framework for Bug Triggering

Let us model the bug triggering process formally. Let:

- P be a deterministic program,
- $\ell_v \in P$ the line of code containing a known vulnerability,
- $i \in I$ an input to P ,
- $E(P, i)$ the execution trace of P on input i .

We define the bug-triggering predicate as:

$$T_{\ell_v}(i) \iff E(P, i) \text{ triggers the bug at } \ell_v$$

For this predicate to evaluate to true, three conditions must be satisfied:

1. **Reachability**: the execution must reach the vulnerable line ℓ_v ,
2. **Path Validity**: the execution must follow a control-flow path that permits the bug to trigger,
3. **State Satisfaction**: the program must be in an appropriate internal state at ℓ_v for the bug to manifest.

We define each of these as a logical constraint on the input i , decomposing T_{ℓ_v} into increasingly specific sets of predicates:

Reachability Constraints (F_r). Let Z_{ℓ_v} be the set of all control-flow paths that reach ℓ_v . Each path $p \in Z_{\ell_v}$ defines a path predicate PC_p , which is the conjunction of all branch conditions along the path:

$$PC_p = \bigwedge_{bb \in p} c_{bb}$$

Then, the reachability condition is:

$$F_r = \bigvee_{p \in Z_{\ell_v}} PC_p$$

Triggering Path Constraints (F_p). Only a subset of reachable paths actually allows the bug to trigger. Let $Z_v \subseteq Z_{\ell_v}$ denote this subset. Then:

$$F_p = \bigvee_{p \in Z_v} PC_p$$

State Constraints (F_s). Even on a valid path $p \in Z_v$, additional semantic conditions may be required to trigger the bug. These constraints are typically independent of control flow and are thus not captured by PC_p . We denote them as SO_p , a conjunction of predicates b_j over the input or program state:

$$SO_p = \bigwedge_j b_j \quad \text{such that } PC_p \not\models SO_p$$

The complete bug-triggering condition is then:

$$T_{\ell_v} = \bigvee_{p \in Z_v} (PC_p \wedge SO_p)$$

This formulation makes explicit the layered nature of bug discovery: reach the vulnerable location, traverse a valid path, and satisfy the necessary state.

```

1 char* copy(char* buffer, int blen) {
2     char *b = calloc(4, sizeof(char));
3     strncpy(b, buffer, blen);
4     /*
5     b1: input[1] > 4
6     b2: strlen(input) > 6
7     */
8     return b;
9 }
10
11 int f(char *input) {
12     char m = input[0];
13     int res;
14     switch (m) {
15         case 'D': // bb15-17: input[0] == "D"
16             res = *((int *) copy(input + 1, 4));
17             break;
18         case 'P': // bb18-21: input[0] == "P"
19             res = atoi(copy(input + 2,
20                             input[1]));
21             break;
22     }
23     return res;
24 }

```

Listing 1: Illustrative Example

5.2.4 Illustrative Example

Consider the simplified C code shown in Listing 1

In this example, a buffer overflow may occur at line 3 (`strncpy`) if the input satisfies certain conditions. To model this:

$$\begin{aligned}
 F_r : & \quad input[0] = 'D' \vee input[0] = 'P' \\
 F_p : & \quad input[0] = 'P' \\
 F_s : & \quad input[0] = 'P' \wedge input[1] > 4 \wedge \text{strlen}(input) > 6
 \end{aligned}$$

This illustrates how each constraint narrows the input space. An input satisfying F_s necessarily satisfies F_p and F_r , but not vice versa. It also demonstrates how path conditions and state conditions can be logically orthogonal.

5.2.5 Bug Complexity Indicators

From this framework, we derive three key complexity dimensions:

- **Reach Complexity:** the difficulty of generating any input satisfying F_r . This reflects how deep or obscure the vulnerable code location is within the program’s control-flow structure.
- **Path Complexity:** the difficulty of producing an input that satisfies F_p , given one that already satisfies F_r . High path complexity arises when only a narrow subset of reachable paths can actually trigger the bug.
- **State Complexity:** the difficulty of satisfying the semantic conditions SO_p , assuming the correct path PC_p is already followed. This represents how sensitive the bug is to the internal state and input values that are not constrained by control flow.

Although these dimensions are layered — each one building on the last — they are semantically distinct and present independent challenges to a fuzzer. For example:

- A bug with high **reach complexity** may reside in code guarded by deep conditionals or rare entry points, making it hard for the fuzzer to even reach the relevant location.
- A bug with high **path complexity** might lie along a path that is reachable but difficult to discover due to tight or conflicting branch conditions.
- A bug with high **state complexity** may be exposed only if specific memory layouts, data dependencies, or value ranges are met — conditions that are invisible to coverage-based guidance.

These insights directly relate to the type of feedback required to detect such bugs. Traditional fuzzers that rely on coverage alone may fail to make progress on F_p or F_s constraints, discarding valuable inputs simply because their execution traces appear “uninformative.”

Prior work by Fioraldi et al. [46] empirically validated this limitation by introducing state-sensitive feedback that enabled the discovery of previously missed bugs. The key takeaway is that even well-covered code may contain undetected vulnerabilities, not because it is unreachable, but because the triggering state is hard to induce.

By explicitly modeling and quantifying these complexity dimensions, we enable a more principled understanding of fuzzing efficacy, and a foundation for constructing more balanced and informative benchmarks.

5.3 Methodology

In this section, we refine our approach for quantifying bug complexity by focusing on an empirical, fuzzing-based measurement framework. As discussed

earlier, bug complexity can be decomposed into three components—reach, path, and state complexity—each representing a distinct challenge in triggering vulnerabilities during fuzzing.

While one could theoretically analyze program code to extract the logical predicates F_r , F_p , and SO_{ℓ_v} and estimate their complexity using constraint solvers, this approach presents significant limitations. Although F_r can be derived from branch conditions, both F_p and SO_{ℓ_v} typically require approximate directed fuzzing solutions that, in practice, reduce back to empirical experimentation. Moreover, constraint solvers often struggle to generate valid inputs in large or complex programs. Given that fuzzing itself is inherently an empirical process, we adopt an experimental approach that uses fuzzing to measure the time required to satisfy each complexity dimension.

Our methodology is implemented through two key components: a lightweight instrumentation layer and a set of four targeted fuzzing campaigns. This design allows us to systematically assess the time needed to discover or trigger vulnerabilities under controlled conditions, thereby estimating the complexity associated with each component.

5.3.1 Program Instrumentation

To enable precise complexity measurement, we extended the default instrumentation provided by coverage-guided fuzzers. In addition to tracking basic block and edge coverage, our system logs the complete execution trace for each input. When the execution reaches a known vulnerability location, we serialize the trace and compute a hash to uniquely identify the path. Each input is stored alongside its trace and outcome (e.g., crash or normal execution), enabling detailed analysis of how fuzzers navigate through vulnerable code and which inputs cause specific behaviors.

5.3.2 Targeted Fuzzing Campaigns

We designed four complementary fuzzing campaigns to isolate and quantify each bug complexity component. Each campaign is repeated five times to account for the inherent randomness of fuzzing.

Campaign 1 (C1): Reachability Complexity

The goal of the first campaign is to establish a baseline measurement for *reachability*. Using the default seed corpus provided by the target software, the fuzzer runs for five days, measuring the time required to reach the vulnerable code location for the first time—regardless of whether the vulnerability is triggered. All inputs that reach the target location are collected and stored for subsequent campaigns. The primary metric extracted from this campaign is the *Time-To-Reach (TTR)*, which directly reflects the difficulty of satisfying the reachability predicate F_r .

Campaign 2 (C2): Path Exploration

The second campaign aims to increase *path diversity* to the vulnerable location. It uses a seed corpus composed exclusively of inputs that previously triggered a crash. A targeted fuzzer—rewarded for retriggering the bug while covering new edges—is run for 12 hours to encourage the generation of inputs that explore alternative execution paths, some of which may lead to the vulnerability even if the bug is not triggered. After the campaign, we analyze the collected execution traces to identify *potentially vulnerable paths*—those that reach the vulnerable code but do not always trigger the bug—thus approximating the Z_v set introduced in our formal model.

Campaign 3 (C3): Path Complexity

Campaign 3 is designed to isolate and measure the *path complexity* of a bug—specifically, the difficulty of generating an input that follows a control-flow path necessary to trigger the vulnerability, assuming the vulnerable location is already reachable.

We start this campaign with a seed corpus composed exclusively of inputs that reach the vulnerable code location but do not trigger the vulnerability. By using this targeted corpus, we remove the reachability constraint and focus solely on exploring the subset of control-flow paths that could lead to bug activation.

The fuzzer is executed for five days, mutating the inputs in an attempt to generate inputs that satisfy the path conditions F_p required to activate the bug. Throughout the experiment, we monitor all inputs generated by the fuzzer and record whether they:

1. Trigger the vulnerability directly, or
2. Exercise one of the potentially vulnerable paths identified in Campaign 2.

The key metric collected is the *Time-To-Trigger for Path Complexity* (TTT_{path})—the time elapsed until the first input either triggers the vulnerability or exercises one of the potentially vulnerable paths. By using a corpus that already satisfies reachability, Campaign 3 isolates the challenge of navigating the correct control-flow paths, thereby providing a direct measure of path complexity.

Campaign 4 (C4): State Complexity

The fourth campaign focuses on *state complexity*, which captures the challenge of satisfying the internal program conditions (i.e., $S_{O\ell_v}$) necessary to trigger the vulnerability, given that the correct path has already been traversed. This campaign uses a seed corpus consisting of inputs that follow

potentially vulnerable paths identified in Campaign 2 but do not themselves trigger the vulnerability.

The fuzzer is run for five days, attempting to mutate these inputs into ones that satisfy the hidden state conditions required for bug activation. The metric collected is the *Time-To-Trigger for State Complexity* (TTT_{state})—the time needed for an input following a known vulnerable path to evolve into one that actually triggers the vulnerability. This metric quantifies the effort a fuzzer must expend to satisfy the state conditions, given that path complexity is no longer a limiting factor.

This four-campaign framework enables us to systematically and empirically quantify the reach, path, and state complexities of each vulnerability. By acknowledging the limitations of purely analytical methods and aligning our methodology with the empirical nature of fuzzing, we provide a rigorous yet practical approach to understanding and improving fuzzing efficacy in real-world settings.

5.4 Implementation

To implement our framework, we extended AFL++ [47], a state-of-the-art mutation-based, coverage-guided fuzzer. Our implementation relies on Clang’s `trace-pc-guard` instrumentation to record all the covered edges. We also extended the fuzzer runtime to monitor, for each input, whether the vulnerable code line is executed and to store the complete execution trace in shared memory.

Upon completion of each test case, the fuzzer checks whether the input has reached the known vulnerability. If so, it hashes the corresponding execution trace and updates a map that tracks, for each path hash, the number of crashing and non-crashing inputs that have been observed.

To align with the methodology described in Section 5.3, we patched AFL++’s feedback system in two main ways:

- In Campaign 2, the feedback logic retains non-crashing inputs that reach the vulnerable code line. This allows us to steer the fuzzer toward diverse executions that still interact with the bug location, reducing the bias of crash exploration mode towards only crashing inputs.
- In Campaign 4, we further constrained the fuzzer to consider only inputs that reach the vulnerable code line. This ensures that the search focuses on satisfying the state constraints (F_s) without exploring unrelated code paths that could increase the time-to-crash without being indicative of how actually hard it was reach the desired state.

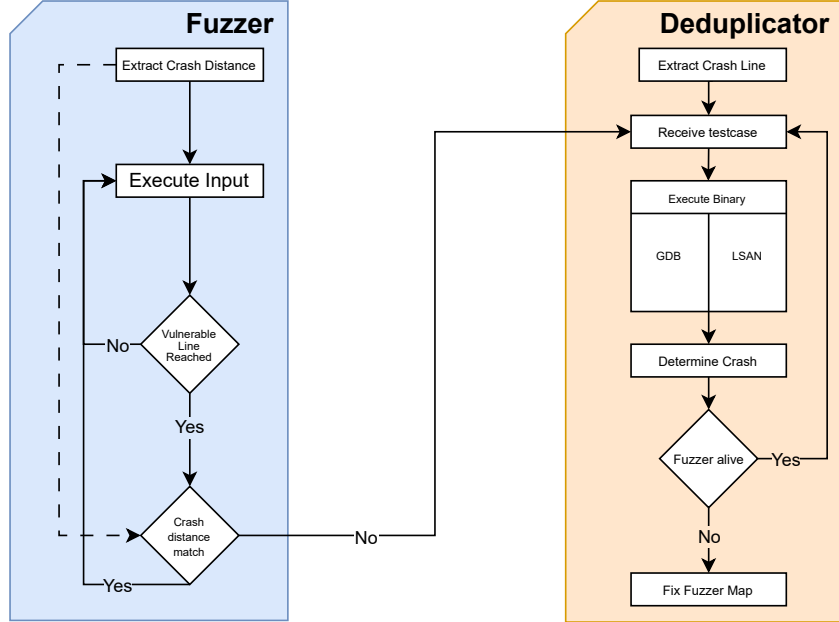


Figure 5.1: Crash Deduplication Architecture

5.4.1 Bug Reproduction and Deduplication

A problem we encountered in our first experiments with our framework is that the presence of other bugs could pollute the results in unpredictable ways. For instance, when exploring different paths to reach a vulnerability, the fuzzer could make the target application crash for other reasons (e.g., another unknown bug), making us conclude that the original vulnerability is easier to trigger than it actually is.

In this context, we required a runtime bug deduplication mechanism, but the crash deduplication system in AFL++ did not meet our needs. Specifically, AFL++’s built-in deduplication differentiates inputs based on the program’s covered edges, which means it considers the execution path taken by the input. However, we are primarily interested in determining whether a crash is simply a retrigger of the same underlying bug, regardless of the path taken to reach it. Additionally, other approaches that rely on hashing the stack trace at the crash location—such as honggfuzz’s deduplication system—suffer from the same limitation: they are too fine-grained for our purposes, splitting similar crashes that stem from the same root cause into separate groups.

Therefore, to determine whether a crash corresponds to the vulnerability under analysis, we developed a two-phase de-duplication system, summa-

rized in Figure 5.1.

The first phase is a lightweight online check. For each crashing input that reaches the vulnerable line, we compute the distance between the last occurrence of the vulnerable line in the execution trace and the crash location (i.e., the last edge). If this distance matches exactly the one observed in the original reproducer input, we consider the bug as reproduced. This decision is justified by our use of trap-on-error sanitizers (in OSS-Fuzz) and lethal canaries (in MAGMA), which terminate the program almost immediately upon bug activation. As a result, we observe consistent “line-to-crash” distances across multiple triggering inputs.

However, this heuristic may fail in some cases—especially when sanitizer-induced failure handling introduces variable-length trace patterns, making the previous condition sufficient but not necessary. In other words, there might be cases in which the distance is different from the expected one, but the crash is still a consequence of the same vulnerability.

To account for this, we introduce a second verification phase. In this second phase, we rerun the crashing input under `gdb` to collect a backtrace. We then filter out all error-handling routines (e.g., `asan` internals or `magma` logging utilities), and compare the code line of the first non-filtered frame against the one obtained from the original reproducer. If they match, we consider the crash as a valid retrigger; otherwise, the input is discarded.

Due to compatibility limitations of `LeakSanitizer` with `gdb`, we apply this method only to targets that support GDB-based trace collection. For targets like `git_21886` and `fluent_bit_26345`, we instead rely on the crash backtrace extracted from the sanitizer logs.

To minimize the impact of crash deduplication on our fuzzing campaign’s resource utilization, we ran the deduplicator on a separate machine. A lightweight local crash sender was responsible for collecting the fuzzer inputs, communicating them to the remote deduplicator, relaying the deduplication results back to the fuzzer, and updating the hash map once the campaign concluded.

This two-phase deduplication pipeline enables us to maintain high precision in bug reproduction, ensuring that only genuine retriggers are counted in the complexity metrics reported in Section 5.5 while reducing the impact on the fuzzing performance since the only blocking part is the lightweight distance check while the expensive deduplication process runs in parallel to the campaign itself.

5.5 Experiments

In the previous sections, we introduced our model of bug complexity, and we described how we can estimate its three components by measuring the time a fuzzer takes to generate inputs that satisfy the $\mathcal{F}_r$, $\mathcal{F}_p$, $\mathcal{F}_s$ predicates.

Bug	Reach Complexity	Path Complexity	State Complexity
PNG001	-	Low	Very High
PNG003	-	Low	Low
PNG006	-	Very High	-
PNG007	-	High*	-
TIF001	High*	Very High	-
TIF002	High*	Medium*	Low
TIF007	Low	Low	Low
TIF008	Very High*	Very High	-
TIF012	-	Medium	-
TIF014	Low	Medium	Low
SQL003	High*	Very High	-
SQL010	Low	Very High	-
SQL015	Low	Very High	-
SQL018	Low	Very High	-
SQL020	High*	Low	-
XML001	-	Very High	-
XML003	-	Very High*	-
XML009	-	High*	-
XML012	-	Very High	-
XML017	-	Low	-
SSL001	-	Medium	-
PDF001	-	High*	-
PDF002	-	Very High	-
PDF010	-	Medium	-
PDF011	-	Very High	-
PDF016	-	Medium	-
PDF018	High*	Medium*	-
PDF019	-	High*	-
PDF021	-	Very High	-

Table 5.1: Complexity metrics for bugs contained in MAGMA Benchmark.

As a reminder, by starting from the initial corpus, we use the time required to generate the first input that reaches the bug location as a proxy for the *reach complexity*. Then, starting only from seeds that already reach the location, we measure the time the fuzzer takes to generate an input that follows one of the paths required to trigger the bug, and use this as a measure of *path complexity*. Finally, we run a third and final fuzzing campaign, this time providing only seeds that already follow the required paths, and observe the time required to actually trigger the bug, capturing the *state complexity*. Since we repeat every experiment five times to account for the non-deterministic nature of fuzz testing, to compute the complexity of a single vulnerability, we performed roughly **75 days of fuzzing** (three campaigns of five days each, all repeated five times).

To simplify the presentation, in the rest of the section we cluster the time measurements into four classes: **Low** when the time required to complete the task is less than three hours, **Medium** when it is between 3 and 24 hours, **High** when it is between 1 and 5 days, and **Very High** if the fuzzer was never able to satisfy the condition in our experiments. In the rest of the section we explore the prevalence of the various complexity classes across the two most popular datasets that are used today to evaluate the performance of fuzzers: MAGMA [56] and OSS-FUZZ [19].

5.5.1 MAGMA

For our first test, we evaluate a selection of bugs from the MAGMA [56] fuzzing benchmark. First introduced in 2020, this is still one of the most recent and most popular datasets used to evaluate and compare fuzzers. Contrary to other benchmarks that relied on artificial vulnerabilities, such as LAVA [42], the purpose of MAGMA was to re-use a selection of real-world vulnerabilities. For our experiments we excluded bugs that did not have a reproducing test case, since this is a necessary condition for performing our analyses, as described in section 5.3. We also excluded the PHP target, which did not build in our testing environment. As a result, in our first test we evaluated the complexity of 29 vulnerabilities present in the MAGMA benchmark. As an initial corpus for the reachability campaign we used the standard corpora provided by magma for the different projects.

Our analysis of the MAGMA-based dataset is summarized in Table 5.1. Each result represents the average across the five runs, and it is marked with an asterisk (*) if the complexity class varied between different runs. Thus, the first observation is that our measurements were remarkably stable: with only 12 of the 87 measurements showing any variation across the five experiments.

By looking at the three components, we can observe that the *reach complexity* does not appear to be an issue for the 29 vulnerabilities we measured. In fact, our results show that over 75% of the bugs have a low reach complexity, with 18 of them being reached directly with the initial corpus. Among the remaining 25%, only three bugs exhibited consistently high scores across all five runs, indicating persistent reachability issues.

Path complexity emerged instead as the most prominent challenge for our fuzzer. Specifically 45% of the bugs (13 out of 29) received the maximum path complexity score, in twelve cases with zero variance. This means that our fuzzer was not able to find the right path in any of the 5-day experiments, even when starting from a corpus that only included inputs that already reached the vulnerable location. Among the remaining 16 bugs, only five had consistent low path complexity across the five runs, with the rest falling into the medium or high categories. The PNG007 vulnerability was one of the most prominent cases of high variance, with one campaign in which

Bug	Discovery Days	Reach Complexity	Path Complexity	State Complexity
json_342	0	Low	Low	Low*
json_343	0	Low	Low	Low
json_344	0	-	Low	Low
git_21836	0	-	Medium*	-
serenity_28886	0	-	Very High	Low
fluent_bit_27279	0	-	Low	-
fluent_bit_26345	0	-	Low	-
cblosc2_26508	0	Low	Low	Low
libxml2_25067	0	-	Very High	-
simdjson_26858	0	Low	Very High	-

Table 5.2: Complexity and time-to-discovery of real-world bugs from OSS-Fuzz (fast discovery set).

Bug	Discovery Days	Reach Complexity	Path Complexity	State Complexity
llvm_28856	1210	Very High	Low*	-
gdal_26596	1215	Very High	Medium*	-
gdal_14512	716	Very High	Low	-
tpm2_23127	1123	-	Low	-
poppler_25631	845	-	High*	-
pcre2_14376	901	Medium	Very High	-
wxwidgets_20527	717	Low	Very High	Low
libarchive_14689	901	Medium	High*	-
libwebp	ND	-	Very High	Low
cblosc2_26442	106	Very High*	Medium*	-

Table 5.3: Complexity and time-to-discovery of real-world bugs from OSS-Fuzz (slow discovery set).

the fuzzer discovered the right path in under three hours, while two others failed to do so even after five days. These discrepancies in path complexity may be due to input shadowing, a phenomenon previously documented in the literature [86], where promising inputs are discarded due to a lack of perceived novelty.

Finally, the most interesting observation about our experiments is that *state complexity* is minimal across the MAGMA dataset we tested. Only 17% of the bugs exhibited any sign of state complexity, and only one showed a significant state-related difficulty. This suggests that subtle runtime conditions are rarely the dominant obstacle for this dataset.

5.5.2 OSS-FUZZ

As a second example of the application of our taxonomy, we looked at different real-world bugs taken from OSS-FUZZ [19]. In this case, we selected twenty vulnerabilities, divided into two main groups: the first one (*fast discovery set*) contains bugs with a very low time-to-discovery, whereas the second (*slow discovery set*) contains bugs that required a large amount of time to be discovered, as noted by Zhu et al. [112]. This distinction can help us verify whether our complexity metrics correlate with empirical data, thus expecting a higher time to discover to be indicative of a higher complexity

and vice versa. When available, as an initial corpus, we used the corpus automatically generated via the ossfuzz scripts. Otherwise, we used the test cases provided with the target project. We avoided using the public corpora provided by OSS-Fuzz because, given their high saturation, they would have introduced a large bias in the measurement of complexity. The selection criteria for these bugs were as follows. First, we considered the discovery delay reported by OSS-Fuzz and selected bugs found within the first day for the fast-discovery set, and bugs discovered only after more than three months for the slow-discovery set. Second, we required the availability of a reproducer test case for each bug, since this is necessary for Campaign 2. Finally, to avoid over-representing any single codebase and biasing the results toward one project’s characteristics, we limited the selection to at most three bugs per project.

The results of the tests performed on OSS-FUZZ are summarized in Table 5.2, alongside the number of days required for OSS-FUZZ to discover the bug (extracted from [112]). As a first observation, we can notice that indeed the time-to-discovery correlates with our measure of bug complexity. Among the fast discovery set, six out of ten vulnerabilities present no complexity at all, and one (`git_21836`) has a medium path complexity of 5 – indicating that in the worst case the right path was discovered in less than 6 hours of fuzzing. The remaining three cases were all trivial to reach, but presented very high path complexity. In the slow discovery, we observed the opposite trend, with 7 out of 10 bugs presenting a very high complexity, and not a single bug with low complexity overall.

Similarly to what we observed in the MAGMA benchmark, also in this case the most prominent source of complexity is related to the path. However, in this dataset 40% of the slow-discovery bugs showed a very high *reach complexity*, which was instead almost always absent in the MAGMA experiments. On the other hand, all fast-discovery bugs were located in parts of the program that were trivial to reach.

Finally, while the overall number of bugs with a state-dependent component is exactly doubled with respect to MAGMA (35% vs 17%), the complexity of the state predicates remained very low and our fuzzer was able to satisfy them easily in all experiments.

5.5.3 Edge dependency

So far we observed that the main obstacle for a fuzzer to discover the vulnerabilities we tested (both in the case of MAGMA and OSS-FUZZ) is the complexity of reaching the vulnerable line with the right control-flow path. We now look at this aspect in more detail, in particular from the angle of a coverage-based graybox fuzzer, given their prevalence in real world fuzzing contexts.

Coverage-guided fuzzers consider inputs interesting only if they increase

a coverage metric, usually related to the discovery of new edges. Consequently, if a code region is saturated via non-crashing inputs that already explore all edges, the fuzzing process will end up de-prioritizing the exploration of this area, shifting its focus elsewhere. To evaluate whether this could be the cause of a high path complexity, we analyzed all paths explored by the fuzzer and computed an additional metric: *Edge Dependency*. We consider a bug to be edge-dependent if there exists at least one edge that discriminates between crashing and non-crashing paths. In other words, when a bug is *not* edge-dependent it means that a fuzzer could cover all edges and still not find the right path required to trigger a vulnerability.

To assess this property in our dataset we re-executed all the inputs saved during Campaigns C2 and C3. Instead of hashing the execution traces, we kept the full list of edges covered by each input. Once this data was collected, we separated the traces corresponding to the vulnerable and potentially vulnerable paths from the rest. We then checked whether or not these sets contained any edge absent from the non-crashing traces.

The results of our analysis are reported in Table 5.4. Among the bugs for which our fuzzer was able to discover the required path, 84.62% exhibited edge dependency. In these cases, the reward provided to our fuzzer (designed to increase edge coverage) would eventually lead to the discovery of the right path. In contrast, only 47.06% of the bugs we could not retrigger over 5 days of fuzzing showed this property. This confirms that edge-dependency is one factor that affects path complexity. But it is not the only one. In fact, numerous bugs had a very high complexity while still presenting a discriminant edge. To better understand these cases, we looked at the absolute number of paths that reach the vulnerable code line and at the proportion of those that are required to trigger the vulnerability (columns 4-6 in Table 5.4).

Again, we can observe a significant contrast in the results: Retriggered bugs exhibit an average of 276K paths with 48.5% of triggering paths. In comparison, the average for non-retriggered bugs is 2.6M paths (an increase of around 10x), but with only 12.32% potentially triggering the vulnerability (4x decrease when compared to the previous class). So the high path complexity in these cases might be a consequence of having fewer “good” paths among many more ‘bad’ ones to explore.

These findings suggest that edge dependency and path sparsity can significantly hinder the testing process for two reasons. When feedback is not sensitive to the vulnerable path or when these paths are needles in a haystack, the fuzzer may drift away from the region containing the fault or fail to mutate inputs capable of activating the bugs, allowing defects to remain undiscovered.

5.6 Discussion

Figure 5.2 shows a summary of how many bugs fall into each complexity class in the three datasets. By looking at these distributions we can make several interesting observations:

Bugs location. Real-world bugs are often located in parts of the program that are difficult to reach, as shown by the results of the OSS-FUZZ Slow Detection dataset in Figure 5.2. However, these cases are under-represented in the MAGMA dataset. At first, one might think that reachability might depend on the size and complexity of the target program. However, we investigated this aspect (all data about program sizes, in terms of raw bytes, basic blocks, and edges on the CFG, are summarized in Table 5.5) but found that no evident correlation exists among binary size and complexity.

State Constraints. Previous works showed that vulnerabilities that require complex state conditions exist, and can be detected by using specific techniques designed for this purpose [46]. However, our results show that these vulnerabilities are very rare in all our datasets. Only one vulnerability out of the 49 we tested had a very high complexity in this category and a stunning 75% of them had no state dependency at all!

A possible explanation is that in the vast majority of the cases, the trigger conditions for bugs are mostly captured by path constraints. If proven true, this would be a fundamental finding not only for fuzzing but for security testing as a whole. On the other hand, this lack of state constraints could also be explained by a bias of how these datasets are constructed. Most datasets are created by assembling vulnerabilities that were detected by fuzzers in the first place, and since today’s fuzzers are not good at exploring state, this might introduce a dangerous positive feedback loop. Which brings us to the next point.

Are Today’s Datasets Fundamentally Biased?

While there is an ongoing debate on whether coverage is the proper metric to compare fuzzers, it is a fact that in reality most of today’s fuzzers are designed to maximize coverage and are evaluated based on this metric. Similarly, the community settled on edge-coverage as the de-facto standard feedback for graybox fuzzing. We made these choices because we observed that they perform well in the average case.

However, our study shows that this last statement may not be correct. In the current methodology, we use fuzzers designed in a certain way to find bugs, and then use these bugs to build a dataset to compare future fuzzers and select which fuzzer works best. This might introduce severe biases that cripple our ability to discover entire classes of bugs. For instance, our measurement of *edge dependence* confirms that when a discriminant edge exists in the vulnerable paths, fuzzers based on edge-coverage are more likely

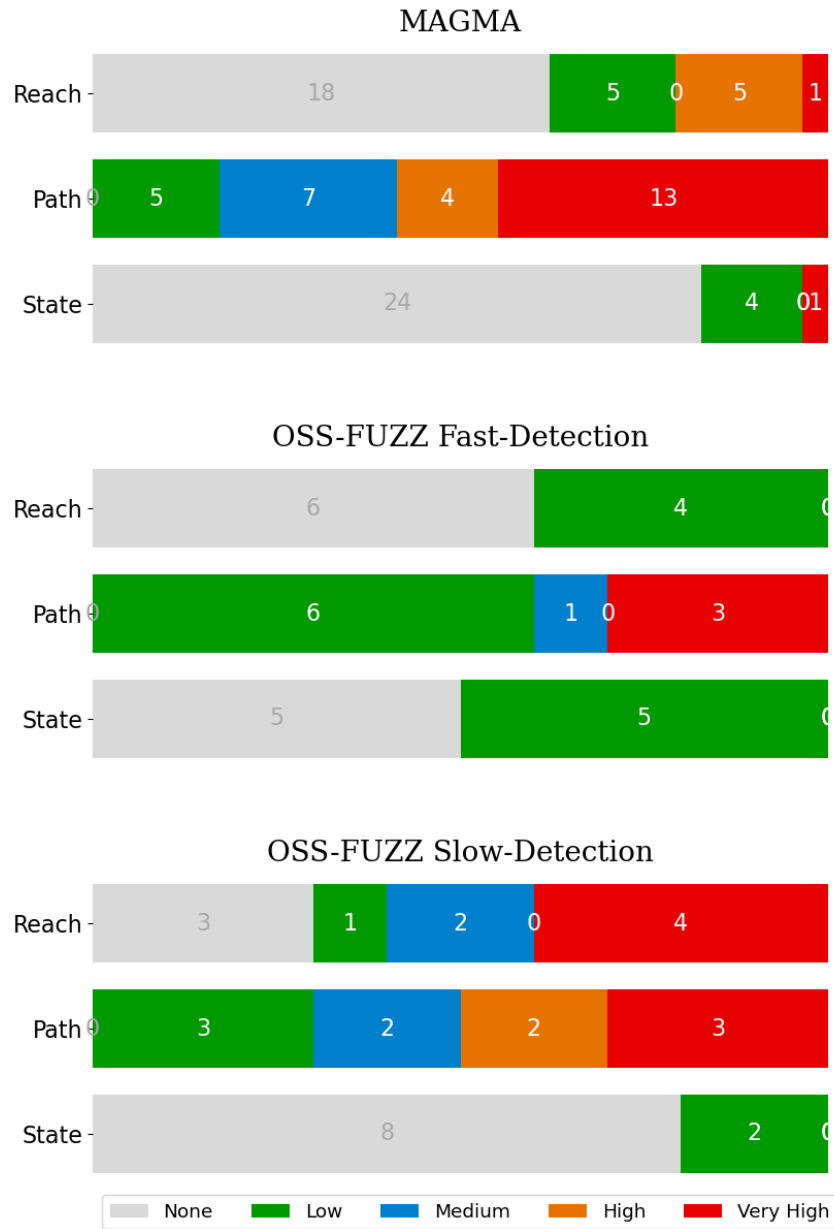


Figure 5.2: Number of Bugs with a given complexity in the three dataset

to discover the bug than when no such edge exists.

More experiments are needed to thoroughly investigate this hypothesis, but we believe our experiments constitute the first step in this interesting direction.

5.7 Limitations & Threats to Validity

In this chapter we present a new way to characterize bugs, which can be divided in two parts. First, we break down the conditions (over the program input) required to trigger a bug in three sets of predicates, required respectively to 1) reach the vulnerable location, 2) follow one of the paths on the CFG associated with the bug, and 3) bring the application into the correct state. Then, in the second part of our characterization, we try to empirically measure the complexity of the three components.

While the presence of the three predicates is a property of the vulnerability, and it is therefore independent from the employed testing technique, their complexity can only be computed with respect to a given technique (e.g., complexity for a given constraint solver or for a particular fuzzer).

Thus, the main limitations of our framework stem from its experimental design, rather than the theoretical model for bug complexity.

Requirements. Our analysis framework has three main requirements. First, to perform the first campaign (C1) we need to know the point in the program where the bug is located. Second, we need at least one test case that can trigger the bug to initiate the second campaign (C2). Finally, to perform the third fuzzing campaign (C3) we need an input that reaches the bug without triggering it (if such input exists).

The first requirement can often be solved by running the crashing input, if any, and analyzing the crash location. The third is usually automatically solved while running the first two campaigns. As a result, the main constraint for our measurement is the availability of a test case that triggers the bug. While this is quite reasonable, such inputs are not always available for all vulnerabilities (for instance, this was not the case for some of the ones in MAGMA, despite its popularity).

Duration of fuzzing campaigns. Another limitation of our approach is the fixed length (set to five days in our experiments) for Campaigns 1, 3, and 4. This threshold may group together bugs with vastly different times-to-reach (in C1) or times-to-crash (in C3 and C4). However, we argue that such a limit is necessary since, in unfavorable scenarios, a campaign may, in principle, run indefinitely without producing useful results.

Five days exceeds the typical budget used in state-of-the-art fuzzing evaluations—often between six hours and two days [59, 69, 50, 25, 112, 24, 29, 100, 49, 31, 105]—by at least a factor of $2\times$. However, ultimately the

choice of the maximum duration is determined by what the analyst would consider *high* complexity.

Randomness of fuzz-testing. Our methodology is also subject to the inherent randomness of fuzzing. Since fuzzing is a stochastic process, good or bad “*luck*” can skew the results in both directions: a complex bug may be discovered early by chance, or, conversely, a simple constraint may be missed due to unfavorable mutation choices.

This issue is most pronounced in Campaign 1, where the corpus is unbiased and the feedback standard. Mutations that do not approach the vulnerable line may still be considered interesting by the fuzzer, potentially leading to drift. A similar problem may arise in Campaign 3, though the bias introduced through corpus selection partially mitigates this risk.

While this randomness may affect the precision of our measurements, its impact is mitigated by averaging results over five independent campaigns. In fact, as we discussed in Section 5.5 in the vast majority of the cases there was no difference in the complexity across the five runs, making us confident that those measurements were not negatively affected by the randomness of the fuzzing process.

Tool selection. The biggest limitations of our study are that, as mentioned above, the actual values we assign to the complexity of each of the three components (reach, path, and state) are estimated experimentally, and therefore depend on the testing technique.

In our case, this means that the values presented in this chapter capture the complexity for a state-of-the-art gray box fuzzer based on edge coverage (AFL++ in our case). As a result, by using a different fuzzer or a different form of coverage and feedback, some of the complexity values might decrease (as the fuzzer requires less time to generate inputs that satisfy the required conditions) or increase. An example can be a reduction of reach, and potentially path, complexity when the fuzzer leverages direct systems since this approaches prioritize reaching a specific area of decode instead of exploring most of the regions as it is the case with coverage-guided fuzzers.

However, this does not affect our main results. For instance, the fact that state complexity is almost completely absent in the vulnerabilities we tested from the MAGMA dataset does not depend on the fuzzer. Similarly, the fact that reach complexity is low largely depends on the location of the bug and the initial set of test cases, which again is unlikely to be significantly affected by the testing technique. Finally the computation of edge dependency is mainly influenced by the number and diversity of distinct paths exercised during each campaign. Different fuzzing techniques may explore the path space with varying efficiency, which introduces an indirect bias in how many distinct paths leading to a bug are discovered, even though the underlying edges themselves are a property of the program and remain unchanged. In our case, this bias is mitigated by the combination of corpus selection and

campaign length, which ensured that the fuzzer covered a sufficiently large portion of the reachable path space to make the measured values robust.

Moreover, AFL++ is representative of the most widely adopted class of coverage-guided fuzzers, making it a suitable baseline. In any case, our methodology and our taxonomy remain applicable across any dynamic testing solution, and comparing how different fuzzing techniques affect the measured complexity is an interesting direction for future work.

Applicability in the Embedded Setting The choice of evaluating the framework on desktop-oriented datasets is motivated by two main factors. First, it demonstrates that the challenge of bug complexity is not confined to embedded firmware but represents a general limitation affecting fuzzing effectiveness across domains. Second, it ensures experimental feasibility and scalability, since embedded fuzzing still suffers from severe performance constraints that hinder large-scale campaigns. Nevertheless, the proposed complexity dimensions are defined at the control-flow and semantic levels, and are therefore equally applicable to firmware binaries. When firmware traces or lifted intermediate representations become available, through frameworks such as Avatar2 [74] or PANDA [41], the same measurement process can be directly applied to embedded targets, confirming the domain-agnostic nature of the approach.

5.8 Conclusion

In this work, we introduced a novel taxonomy for bug complexity in fuzzing, along with an empirical methodology for quantifying it. Our aim was to move beyond the simplistic binary classification of bugs as “easy” or “hard,” and instead provide a structured framework that captures the factors that make a bug challenging to trigger through fuzz testing. We identified three orthogonal dimensions of complexity—*reach*, *path*, and *state*—representing, respectively, the difficulty of reaching the vulnerable code location, navigating the required control-flow paths to activate the bug, and satisfying the implicit program state conditions. Applying our methodology to real-world bugs from OSS-Fuzz and the MAGMA benchmark suite, we found that *path complexity* is often the dominant barrier to bug discovery, while *state complexity* appears rare, and *reach complexity* is generally low except in a few cases. Furthermore, we analyzed factors influencing fuzzing difficulty and highlighted two key contributors for coverage-guided greybox fuzzers: the absence of actionable feedback—captured by the *Edge Dependency* property—and the number of distinct paths required to reach the vulnerable code. We conclude that future fuzzing research should treat bug complexity as a first-class evaluation metric, guiding both benchmark design and fuzzer development to better address the real challenges presented by software vul-

nerabilities.

Bug	Path Complexity	Edge Dependency	Total Paths	Crashing Paths	%
PNG001	Low	✓	42,225	26,715	69.26%
PNG003	Low	✓	14,961	13,563	90.65%
PNG006	Very High	✗	870,188	41,966	4.82%
PNG007	High*	✓	24,864	16,815	67.62%
TIF001	Very High	✗	686,870	629,840	91.69%
TIF002	Medium*	✓	241,729	240,574	99.5%
TIF007	Low	✓	127,061	89,506	70.44%
TIF008	Very High	✓	530,492	527,841	99.5%
TIF012	Medium	✓	700,348	350,809	50.09%
TIF014	Medium	✓	332,220	292,763	88.12%
SQL003	Very High	✓	2,027,955	870,557	42.93%
SQL010	Very High	✓	575,712	43,907	7.63%
SQL015	Very High	✓	2,743,625	74,221	2.71%
SQL018	Very High	✓	8,283,277	638,473	7.71%
SQL020	Low	✓	80,419	66,425	82.59%
XML001	Very High	✓	435,247	64,755	14.88%
XML003	Very High*	✓	14,185,039	328,006	2.31%
XML009	High*	✓	41,017	19,959	48.66%
XML012	Very High	✗	196,422	21,051	10.72%
XML017	Low	✓	355,718	353,035	99.25%
SSL001	Medium	✓	31,160	29,172	93.62%
PDF001	High*	✓	449,571	1,074	0.24%
PDF002	Very High	✓	203,547	45,364	22.29%
PDF010	Medium	✓	10,016	3,829	38.23%
PDF011	Very High	✓	676,495	425	0.06%
PDF016	Medium	✓	29,483	10,341	35.07%
PDF018	Medium*	✓	1,717	1,021	59.46%
PDF019	High*	✓	74,924	17,122	22.85%
PDF021	Very High	✓	196,422	21,051	10.72%
json_342	Low	✓	24,711	6,950	28.13%
json_343	Low	✓	142,228	48,944	34.41%
json_344	Low	✓	62,481	12,581	20.14%
git_21836	Medium*	✗	851	65	7.64%
serenity_28886	Very High	✗	47,555	21	0.04%
fluent_bit_27279	Low	✓	12	11	91.67%
fluent_bit_26345	Low	✓	456,958	59,919	13.11%
cblosc2_26508	Low	✗	2435	718	29.49%
libxml2_25067	Very High	✗	2,193,563	6	0.00%
simdjson_26858	Very High	✗	6,990,603	235	0.00%
llvm_28856	Low*	✓	32,427	6510	20.08%
gdal_26596	Medium*	✓	118,812	21,824	18.37%
gdal_14512	Low	✗	385	6	1.56%
tpm2_23127	Low	✗	297	77	25.93%
poppler_25631	High*	✓	3,125,774	31,226	1.00%
pcre2_14376	Very High	✗	3,174,383	40,314	1.27%
wxwidgets_20527	Very High	✗	310,656	1236	0.40%
libarchive_14689	High*	✓	92,906	11	0.01%
libwebp	Very High	✗	339,889	1	0.00%
cblosc2_26442	Medium*	✗	706,215	696	0.10%

Table 5.4: Edge dependency results for our dataset

Bug	# Edges	# Basic blocks	Binary Size
PNG001	179,570	83,802	3.4MB
PNG003	179,557	83,796	3.4MB
PNG006	179,579	83,807	3.4MB
PNG007	179,550	83,795	3.4MB
TIF001	287,407	131,802	6.5MB
TIF002	287,465	131,837	6.5MB
TIF007	287,475	131,839	6.5MB
TIF008	287,454	131,833	6.5MB
TIF012	287,364	131,799	6.5MB
TIF014	287,475	131,843	6.5MB
SQL003	745,360	328,838	18MB
SQL010	683,644	299,019	17MB
SQL015	683,617	299,006	17MB
SQL018	745,366	328,841	18MB
SQL020	683,642	299,017	17MB
XML001	759,792	327,332	18MB
XML003	759,763	327,319	18MB
XML009	759,766	327,321	18MB
XML012	858,562	375,236	20MB
XML017	759,759	327,317	18MB
SSL001	1,225,183	546,940	35MB
PDF001	1,283,359	564,173	42MB
PDF002	1,283,401	564,191	42MB
PDF010	1,283,404	564,196	42MB
PDF011	1,283,393	564,188	42MB
PDF016	1,283,389	564,187	42MB
PDF018	1,283,396	564,186	42MB
PDF019	1,283,402	564,196	42MB
PDF021	1,283,405	564,189	42MB
json_342	117,366	56,271	3.2MB
json_343	117,803	56,506	2.3MB
json_344	117,812	56,506	2.3MB
git_21836*	1,335,519	578,486	36MB
serenity_28886	622,605	326,317	23MB
fluent_bit_27279	657,519	289,090	17MB
fluent_bit_26345	636,545	279,681	17MB
cblosc2_26508	314,083	146,309	13MB
libxml2_25067	828,734	360,588	18MB
simdjson_26858	153,396	72,872	3.5MB
llvm_28856	2,774,906	1,156,030	99MB
gdal_26596*	4,567,123	2,056,174	159MB
gdal_14512	5,671,714	2,401,881	193MB
tpm2_23127	181,246	86,445	5.6MB
poppler_25631*	654,032	294,946	25MB
pcre2_14376	207,601	98,084	3.8MB
wxwidgets_20527	480,775	220,935	20MB
libarchive_14689	545,720	259,438	38M
libwebp	357,585	165,754	7.5MB
cblosc2_26442	799,779	375,935	20MB

Table 5.5: Size Metrics for the Tested Vulnerable Programs

Chapter 6

Conclusion

This dissertation advanced the state of the art in vulnerability *detection* and *testing* for embedded systems along three complementary axes: a low-overhead, remotely attested crash reporting system for spatial memory errors; a principled taxonomy and measurement framework for *bug complexity* in coverage-guided fuzzing; and a high-throughput, in-place fuzzing architecture that scales execution on real hardware. Taken together, these contributions instantiate an *actionable loop* for improving the security of embedded devices. Pre-deployment, E-FUZZEDGE enables scalable, in-place fuzzing on real hardware to uncover and fix vulnerabilities before release. Post-deployment, EMBEDWATCH provides precise reports of failures and exploitation attempts on devices in the field. Those runtime findings then feed back into the bug-complexity framework, which explains per-bug difficulty in terms of *reachability*, *path*, and *state*; the diagnosis informs the next testing cycle by adapting feedback signals, seed corpora, harness resets, and campaign parameters. Iterating this loop aligns lab testing with field reality and progressively raises both pre-release discovery and in-field resilience.

Summary of Contributions and Findings

In light of the overall progression of this dissertation, and following the research questions defined in Section 1.2, this thesis answers them as follows.

RQ1: *How can we accurately assess and report memory safety violations on embedded devices at runtime, despite tight resource constraints and limited observability?*

To answer this question, we introduced a remotely attested, fat-pointer-inspired design that infers object boundaries and reports the precise manifestation site of spatial memory errors across stack, heap, and globals. The protocol tracks definition/use events with minimal trace volume and validates them against a statically derived model. On ARM TrustZone-equipped hardware,

EMBEDWATCH detects both control-flow and data-only violations and localizes their manifestation points with very low runtime cost: overheads ranged from *0.01% to 2.33%* (geometric mean $\approx 0.23\%$), plus $\approx 0.4\%$ when link encryption is enabled. Across synthetic and real vulnerabilities from open-source firmware, the system produced correct crash reports with no observed false positives in our test corpus.

RQ2: *How can the execution throughput of hardware-based fuzzing be improved without relying on peripheral emulation or other unrealistic modelling assumptions?*

To answer this question, we designed and prototyped a one-to-many architecture that decouples input processing from on-device execution and minimizes link traffic by moving coverage analysis onto the device. A lightweight proxy multiplexes multiple input processors to a single executor, keeping the device busy while other processors schedule and analyze inputs. In our evaluation on STM32L0 hardware and a set of embedded targets, this design yielded consistent throughput gains with two input processors (typical speedups $\approx 1.5\times$, up to $2.0\times$), with clear signs of saturation beyond that point due to device and link constraints. These results demonstrate that pragmatic pseudo-parallelism can materially increase executions-per-second for in-place campaigns, without requiring multiple boards.

RQ3: *Which intrinsic properties of real-world vulnerabilities limit the ability of coverage-guided fuzzers to reach or trigger them in practice?*

To answer this question, we first formalized the bug-triggering predicate T_{ℓ_v} and decomposed its difficulty into three orthogonal components that matter to grey-box fuzzing: *reachability* (F_r), *path* (F_p), and *state* (F_s). We then instantiated an experimental methodology, composed of four targeted campaigns with enhanced instrumentation and crash deduplication, to measure each component per bug. Two robust patterns emerged across MAGMA and curated OSS-Fuzz cases: **Path complexity dominates**. Even when the vulnerable line is easily reachable, the correct control-flow to trigger the fault is often elusive and frequently remains unsatisfied over multi-day campaigns.

State complexity is uncommon. Beyond control flow, additional semantic preconditions rarely emerged as the primary blocker in our datasets and, when present, were typically satisfied quickly once F_p was met. Importantly, we caution that this second observation likely reflects a *survivorship bias*: widely used benchmarks and the real findings reported by current engines co-evolve toward bugs that are path-dominant and comparatively state-light, underrepresenting genuinely stateful vulnerabilities. Our interpretation is therefore not “state is unimportant,” but rather “state is under-tested”; this motivates expanding benchmarks with bugs whose triggering

predicates hinge on non-trivial program state (phases, protocol progress, multi-step invariants) and developing guidance that can preserve, model, or learn useful state. We also quantified how *edge dependency* and *path sparsity* shape fuzzing difficulty for coverage-guided engines, highlighting where classic edge-coverage feedback is helpful—and where it is not.

Implications for Practice

For practice, these findings translate into concrete steps. For *deployed devices and test labs*, precise crash reporting with manifestation attribution can be integrated today via compact feedback channels, enabling actionable post-mortem triage without heavyweight tracing. For *fuzzer builders and benchmark curators*, path-sensitive feedback (context/history) should remain a first-class focus, but evaluations should deliberately include stateful bugs to counteract survivorship bias and to incentivize engines that can be sensitive to program state. For *embedded fuzzing workflows*, adopting a one-executor/multi-processor model offers a simple and deployable way to raise executions per second without multiplying boards; operators should expect a device and link-specific saturation point, and tune processor counts accordingly.

Limitations and Future Directions

This work opens several promising avenues for improvement along its three principal axes. Each contribution already demonstrates tangible progress, yet also highlights specific challenges that can inspire future research.

EMBEDWATCH extensions can follow two main directions. The first concerns the remaining limitations that prevent full detection capability for spatial memory errors, such as indirect memory aliasing and tracing interrupt-driven functions. The second, and arguably more intriguing from a research perspective, is the extension of EmbedWatch toward temporal and concurrency-related memory errors. Achieving low-overhead tracking of variable liveness and detecting temporal misuse without a clear pruning criterion remain open problems. Exploring these aspects could lead to new models of runtime analysis that balance detection coverage and performance while maintaining compatibility with resource-constrained devices.

When considering E-FUZZEDGE, the main opportunities for improvement are linked to the communication protocol, which currently fixes the number of parallel input channels. As experiments have shown, larger inputs can slow down device execution, making a high number of generators sub-optimal or even counterproductive. Adapting the number of input processors dynamically—based on live throughput and execution feedback—would maximize overall fuzzing efficiency throughout a campaign. From an engi-

neering standpoint, another natural direction is to integrate E-FuzzEdge with other state-of-the-art on-device fuzzers, such as IPEAFuzz [89], combining their introspection capabilities and execution optimizations. Such integration could enable hybrid, cooperative fuzzing frameworks that leverage multiple devices and testing strategies simultaneously.

Our bug analysis framework also offers fertile ground for future exploration. At present, the system defines its baseline using AFL++; a first research direction would be to evaluate how the proposed complexity metrics vary across different fuzzing infrastructures. This comparison would clarify the respective strengths of diverse fuzzing paradigms and inform the design of testing frameworks that leverage multiple engines. A second, more exploratory direction involves correlating our bug-complexity metrics with static software metrics that can be computed without knowledge of specific vulnerabilities. If meaningful correlations emerge, they could support predictive models that identify code regions most prone to complex bugs. Combining such predictions with adaptive fuzzing configurations would enable more granular testing architectures where different program areas are exercised with specialized mutators, feedbacks, or schedulers to maximize discovery rates.

Collectively, these directions extend this work toward a self-improving ecosystem in which testing, monitoring, and analytical feedback reinforce each other, progressively enhancing the resilience and security of embedded devices.

Outlook

The core message of this thesis is optimistic: with careful problem framing and tight engineering, we can make embedded systems *observable* at runtime, evaluate fuzzers against what actually makes bugs hard, and scale on-device testing without heroic hardware setups. We hope that crash reporting with precise manifestation attribution becomes a standard capability in embedded firmware, bug complexity is adopted as a first-class evaluation lens for future fuzzing research and benchmark curation, and pseudo-parallel in-place fuzzing becomes a practical default for hardware-in-the-loop campaigns. Together, these steps can shorten the path from vulnerability to fix for the billions of embedded devices already shaping our world.

Bibliography

- [1] Arduino project. <https://github.com/mattiasjahnke/arduino-projects>.
- [2] CrashReporting. <https://wiki.ubuntu.com/CrashReporting>.
- [3] EmbedWatch dataset. <https://github.com/laserunimi/embedwatch/tree/master/test>.
- [4] EmbedWatch source code. <https://github.com/laserunimi/embedwatch>.
- [5] Embench: Open Benchmarks for Embedded Platforms. <https://github.com/embench/embench-iot>.
- [6] Fuzzware dataset. <https://github.com/fuzzware-fuzzer/fuzzware-experiments>.
- [7] Intel PT. <https://www.intel.com/content/www/us/en/developer/videos/collecting-processor-trace-in-intel-system-debugger.html?wapkw=intel>.
- [8] Mozilla Crash Reporter. <https://support.mozilla.org/en-US/kb/mozillacrashreporter>.
- [9] Raspberry Pi pico-example. <https://github.com/raspberrypi/pico-examples>.
- [10] SafeStack. <https://clang.llvm.org/docs/SafeStack.html>.
- [11] The Often Misunderstood GEP Instruction. <https://llvm.org/docs/GetElementPtr.html>.
- [12] TI-RTOS Drivers. https://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/tirtos/2_20_01_08/exports/tirtos_full_2_20_01_08/products/tidrivrs_cc13xx_cc26xx_2_20_01_10/docs/doxygen/html/_u_a_r_t_8h.html.
- [13] Iso/iec/ieee systems and software engineering - vocabulary. *IEEE Unapproved Draft Std P24765-2009, Sept 2009*, 2009.

- [14] Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.
- [15] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.
- [16] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, 2019.
- [17] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.
- [18] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
- [19] Abhishek Arya, Oliver Chang, Jonathan Metzman, Kostya Serebryany, and Dongge Liu. Oss-fuzz, 2023. Accessed: 2025-04-23.
- [20] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.
- [21] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.
- [22] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.
- [23] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, June 2006.
- [24] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS ’17*, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

- [25] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, page 1621–1633, New York, NY, USA, 2022. Association for Computing Machinery.
- [26] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043. ACM, 2016.
- [27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.
- [28] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Proceedings of the 36th Annual Computer Security Applications Conference, ACSAC '20*, page 746–759, New York, NY, USA, 2020. Association for Computing Machinery.
- [29] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 18–35, 2022.
- [30] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.
- [31] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. μ FUZZ: Redesign of parallel fuzzing using microservice architecture. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1325–1342, Anaheim, CA, August 2023. USENIX Association.
- [32] Steve Christey, J Kenderdine, J Mazella, and B Miles. Common weakness enumeration. *Mitre Corporation*, 2013.
- [33] Claroty Ltd. The global state of cps security 2024: The business impact of disruptions. Technical report, Claroty Ltd., 2024. Accessed: 2025-11-02.
- [34] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303, 2017.

- [35] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.
- [36] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.
- [37] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. Rept: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 17–32, 2018.
- [38] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.
- [39] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.
- [40] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. *arXiv preprint arXiv:2103.11518*, 2021.
- [41] Brendan Dolan-Gavitt, Joshua Hodosh, Patrick Hulin, Engin Kirda, Tim Leek, and Wenke Lee. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW 2015)*. ACM, 2015.
- [42] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. 05 2016.
- [43] Yubo Du, Yanan Guo, Youtao Zhang, and Jun Yang. Rtt-uaf: Reuse time tracking for use-after-free detection. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS '24*, page 376–387, New York, NY, USA, 2024. Association for Computing Machinery.

- [44] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, volume 17, pages 1–15, 2017.
- [45] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254. USENIX Association, August 2020.
- [46] Andrea Fioraldi, Daniele Cono D’Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.
- [47] Andrea Fioraldi, Dominik Maier, Heiko Ei feldt, and Marc Heuse. Afl++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies, WOOT’20*, USA, 2020. USENIX Association.
- [48] Andrea Fioraldi, Dominik Maier, Heiko Ei feldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *13th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020.
- [49] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS ’22*, page 1051–1065, New York, NY, USA, 2022. Association for Computing Machinery.
- [50] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.
- [51] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.
- [52] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Dangzero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS ’22*, page 1307–1322, New York, NY, USA, 2022. Association for Computing Machinery.

- [53] Michael Grottke, Michael Grottke, Kishor S. Trivedi, and Kishor S. Trivedi. Software faults, software aging and software rejuvenation(new development of software reliability engineering). *null*, 2005.
- [54] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. Deepvsa: Facilitating value-set analysis with deep learning for post-mortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1787–1804, 2019.
- [55] Eric Gustafson, Marius Muench, Chad Spensky, , Nilo Redini, Aravind Machiry, Aurelien Francillon, Davide Balzarotti, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. 2019.
- [56] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.
- [57] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM Conference on Computer and Communications Security*, 2021.
- [58] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. FreeWill: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2497–2512, Boston, MA, August 2022. USENIX Association.
- [59] Adrian Herrera, Mathias Payer, and Antony L. Hosking. `jscpidataflowi/scpii` : Toward a data-flow-guided fuzzer. *ACM Transactions on Software Engineering and Methodology*, 2023.
- [60] ARM Holdings. Arm security technology: Building a secure system using trustzone technology. *Retrieved on June, 10:2021*, 2009.
- [61] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.
- [62] PK Kapur, AK Bardhan, and S Kumar. On categorization of errors in a software. *Intrenational Journal of Management and System*, 16(1):37–38, 2000.
- [63] PK Kapur, S Younes, and S Agarwala. Generalized erlang software reliability growth model. *ASOR Bulletin*, 14(1):5–11, 1995.

- [64] Sujata Khatri, Sujata Khatri, Rajender Singh Chhillar, Rajender Singh Chhillar, Rajender Singh Chhillar, Vir Singh, V. B. Singh, V. B. Singh, V. B. Singh, and Vikramjeet Singh. Measuring bug complexity in object oriented software system. *ACM Sigsoft Software Engineering Notes*, 2011.
- [65] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. Securing Real-Time microcontroller systems through customized memory view switching. In *NDSS*, 2018.
- [66] Tim Kornau. Return oriented programming for the arm architecture. In *Ph.D thesis*.
- [67] Yinxi Liu and Wei Meng. Dsfuzz: Detecting deep state bugs with dependent state exploration. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23*, page 1242–1256, New York, NY, USA, 2023. Association for Computing Machinery.
- [68] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.
- [69] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with data dependency information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 286–302, 2022.
- [70] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.
- [71] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy, S&P/Oakland'21*, May 2021.
- [72] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.
- [73] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. Pomp++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering*, 47(9):1929–1942, 2019.

- [74] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar²: A multi-target orchestration platform. In ISOC, editor, *BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA*, San Diego, 2018. © ISOC. Personal use of this material is permitted. The definitive version of this paper was published in BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA and is available at [:](#).
- [75] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, 2 edition, 2004.
- [76] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.
- [77] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, jun 2009.
- [78] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.
- [79] Zhenyu Ning and Fengwei Zhang. Ninja: Towards transparent tracing and debugging on {ARM}. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 33–49, 2017.
- [80] PSA Certified. Confidence to create: The iot industry action plan to reduce the cost of security. Technical report, Arm Limited, 2022. Accessed: 2025-11-02.
- [81] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, Lihua Guo, and Zhenyu Chen. Funfuzz: Greybox fuzzing with function significance. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–34, 2025.
- [82] Zachary B. Ratliff, Zachary B. Ratliff, Zachary B. Ratliff, D. Richard Kuhn, D. Richard Kuhn, Raghu N. Kacker, Raghu N. Kacker, Yu Lei, Yu Lei, Kishor S. Trivedi, and Kishor S. Trivedi. The relationship between software bug type and number of factors involved in failures. *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2016.

- [83] Takamichi Saito, Ryohei Watanabe, Shuta Kondo, Shota Sugawara, and Masahiro Yokoyama. A survey of prevention/mitigation against memory corruption attacks. In *2016 19th International Conference on Network-Based Information Systems (NBIS)*, pages 500–505. IEEE, 2016.
- [84] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256, Boston, MA, August 2022. USENIX Association.
- [85] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256, Boston, MA, August 2022. USENIX Association.
- [86] Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. Novelty not found: Adaptive fuzzer restarts to improve input space coverage (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, FUZZING 2023, page 12–20, New York, NY, USA, 2023. Association for Computing Machinery.
- [87] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, page 28, USA, 2012. USENIX Association.
- [88] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.
- [89] Jiameng Shi, Wenqiang Li, Wenwen Wang, and Le Guan. Facilitating non-intrusive in-vivo firmware testing with stateless instrumentation.
- [90] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

- [91] V. B. Singh, P. K. Kapur, and Abhishek Tandon. Measuring reliability growth of software by considering fault dependency, debugging time lag functions and irregular fluctuation. *SIGSOFT Softw. Eng. Notes*, 35(3):1–11, May 2010.
- [92] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.
- [93] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.
- [94] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.
- [95] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. {ScaRR}: Scalable runtime remote attestation for complex systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 121–134, 2019.
- [96] Flavio Toffalini, Mathias Payer, Jianying Zhou, and Lorenzo Cavallaro. Designing a provenance analysis for sgx enclaves. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 102–116, 2022.
- [97] Lionel Sujay Vailshery. Internet of things (iot) connected devices installed base worldwide from 2020 to 2030. <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>, 2024. Accessed: 2025-08-12.
- [98] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, Chaoyang District, Beijing, September 2019. USENIX Association.
- [99] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free attacks with fast forward allocation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2453–2470. USENIX Association, August 2021.
- [100] Hang Xu, Liheng Chen, Shuitao Gan, Chao Zhang, Zheming Li, Jianggan Ji, Baojian Chen, and Fan Hu. Graphuzz: Data-driven seed scheduling for coverage-guided greybox fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 33(7), August 2024.

- [101] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 529–540, 2016.
- [102] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 17–32, 2017.
- [103] Nikita Yadav and Vinod Ganapathy. Whole-program control-flow path attestation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23*, page 2680–2694, New York, NY, USA, 2023. Association for Computing Machinery.
- [104] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Parichack: An efficient pointer arithmetic checker for c programs. In *5th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2010.
- [105] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, August 2018. USENIX Association.
- [106] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391. IEEE, 2017.
- [107] Dongjia Zhang, Andrea Fioraldi, and Davide Balzarotti. On Understanding and Forecasting Fuzzers Performance with Static Analysis. In *Proceedings of the 31th ACM conference on Computer and communications security (CCS)*, CCS '24. ACM, October 2024.
- [108] Hang Zhang, Jangha Kim, Chuhong Yuan, Zhiyun Qian, and Taesoo Kim. Statically discover cross-entry use-after-free vulnerabilities in the linux kernel.
- [109] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4345–4363, Boston, MA, August 2022. USENIX Association.
- [110] Yumei Zhang, Xinzhi Liu, Cong Sun, Dongrui Zeng, Gang Tan, Xiao Kan, and Siqi Ma. Recfa: resilient control-flow attestation. In *Annual Computer Security Applications Conference*, pages 311–322, 2021.

- [111] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: State-Aware symbolic execution for exploring State-Dependent program paths. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4365–4382, Boston, MA, August 2022. USENIX Association.
- [112] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21*, page 2169–2182, New York, NY, USA, 2021. Association for Computing Machinery.