Fuzzing Techniques for Automated Vulnerability Detection in IoT Firmware

Kai Feng

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE & ENGINEERING

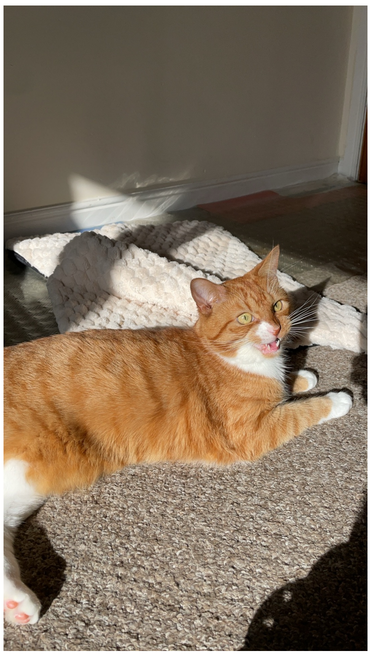
OCTOBER 2025

To my friends,

my parents,

My girlfriend,

My supervisor,
and all my cats



Abstract

A security flaw in the firmware of microcontrollers (MCUs) can lead to devastating consequences. Finding and fixing these bugs before deployment is essential because patching them in the field is often difficult, expensive, or impossible. However, standard software testing techniques like fuzzing struggle with embedded firmware due to its tight coupling with specialized hardware, which makes testing slow, inaccurate, and inefficient. This thesis studies two key design choices: where tests run (emulation, Hardware-in-the-Loop (HIL), or on-device) and what feedback and inputs they use (control flow vs. data flow; generic vs. domain-specific). It moves testing from slow emulation to real hardware and replaces simple code coverage with data-flow guidance to drive bug finding. It also measures how new hardware features can prevent whole classes of bugs.

The approach is demonstrated through four linked contributions. First, Sizzler solves the input wasted problem by generating valid, domain-aware tests for Programmable Logic Controllers (PLCs) by deep learning model, so fuzzing effort is not wasted. Second, FuzzRDUCC improves feedback by tracking def-use chains, revealing subtle bugs that edge-based coverage can miss. Third, Hardfuzz brings this data-flow guidance onto real hardware, using hardware breakpoints for fast, consistent testing. Finally, a differential testing framework for MicroPython compares builds with and without architectural memory-safety features from CHERI and shows which bug classes they block.

These results show that firmware testing benefits from hardware-centric, data-flow-guided methods. These approaches yield smarter, domain-aware inputs; feedback that is more informative than edge coverage; and fast, consistent testing on real devices. It also provides clear evidence that architectural memory safety-exemplified by CHERI-can block whole classes of vulnerabilities. In short, the thesis shifts the goal from only finding bugs to also preventing them by design.

Contents

List of Tables

List of Figures

Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution

Kai Feng, 冯凯

Abbreviations

- `AFL` - American Fuzzy Lop
- `AP` - Architectural Permissions
- `ARM ETM` - ARM Embedded Trace Macrocell
- `CFGs` - Control Flow Graphs
- `CHERI` - Capability Hardware Enhanced RISC Instructions
- `CST` - Concrete Syntax Tree
- `CVE` - Common Vulnerabilities and Exposures
- `def-use chain` - definition-use chain
- `DDC` - Default Data Capability
- `DMA` - DIRECT MEMORY ACCESS
- `DUT` - Device Under Test
- `EBB` - Extended Basic Block
- `GAN` - Generative Adversarial Network
- `GPIO` - General-Purpose Input/Output
- `HiL` - Hardware-in-the-Loop
- `HMI` - Human Machine Interface
- `ICS` - Industrial Control Systems
- `Intel PT` - Intel Processor Trace
- `IoT` - Internet-of-Things
- `IR` - Intermediate Representation
- `JTAG` - Joint Test Action Group

- `LD` - Ladder Diagram
- `LLM` - Large Language Model
- `LSTM` - Long Short-Term Memory
- `MCUs` - Microcontroller Units
- `MLE` - Maximum Likelihood Estimation
- `MMIO` - Memory-Mapped Input/Output
- `PC` - Program Counter
- `PCC` - Program Counter Capability
- `PLC` - Programmable Logic Controller
- `PoC` - Proof of Concept
- `PUT` - Program Under Test
- `QEMU` - Quick Emulator
- `RG/PG` - Programmiergerät/Personal Computer
- `SeqGAN` - Sequential Generative Adversarial Network
- `SDP` - Software-Defined Permissions
- `SoCs` - System-on-Chips
- `SUT` - System Under Test
- `SWD` - Serial Wire Debug
- `TBB` - Translation Block

Chapter 1

Introduction

The proliferation of Internet-of-Things (IoT) devices, powered by microcontroller units (MCUs), has reshaped the landscape of modern computing. These small, low-cost systems-on-chip (SoCs) sit at the heart of embedded systems [1]. Beyond consumer devices such as thermostats and wearables, MCUs are central to industrial control systems (ICS) and critical infrastructure. Programmable Logic Controllers (PLCs)-specialized MCU-based controllers-run automated processes in power grids, water treatment, and smart factories.

While this technological integration drives efficiency, it also introduces severe security risks. A vulnerability in a consumer device might lead to data leakage, however, a flaw in a PLC can have far more devastating consequences. A prominent example is the class of vulnerabilities affecting widely-used PLCs such as the Siemens S7 family. A specific vulnerability, CVE-2022-38465, could allow attackers to discover the private key of a CPU product family by an offline attack against a single CPU member of the family[1]. Attackers could then use this knowledge to extract confidential configuration data from projects that are protected by that key or to perform attacks against legacy Programmiergerät/ Personal Computer (PG/PC) and Human-Machine Interface (HMI) communication. The compromise of such systems can lead to catastrophic outcomes, including widespread power outages, disruption of essential services, and significant threats to public safety.

1. Further details available at: https://nvd.nist.gov/vuln/detail/CVE-2022-38465

Because these attacks often exploit firmware bugs, *finding and fixing vulnerabilities before deployment* is essential. Post-deployment patching is slow, costly, or infeasible in industrial settings. A notable example of an unpatchable hardware-related flaw is the bootrom vulnerability in the Nintendo Switch's SoC, which permanently compromised the system's security[2]. In critical infrastructure, patching may require a complete shutdown and reboot of physical systems, leading to high financial and operational costs. Consequently, it is imperative for developers to identify and remediate firmware bugs at the source-code level through rigorous in-house testing. To this end, extensive security testing during the development phase is the most effective strategy to prevent or mitigate such attacks.

Researchers have explored vulnerability detection techniques including remote attestation [2], compartmentalization [3, 4], and static and dynamic analysis [5, 6, 7]. In practice, dynamic methods are attractive to practitioners: they reduce false positives, add no runtime cost to the final product, and can find exploitable bugs before release.

Fuzz testing, or fuzzing [8], has emerged as one of the most successful dynamic analysis techniques for discovering security flaws. Fuzzing operates by generating a multitude of test cases to repeatedly execute a target program while monitoring for exceptions, such as crashes or timeouts, which indicate potential vulnerabilities. A typical fuzzer maintains a queue of ”seeds”—inputs known to be interesting—and iteratively mutates them to generate new test cases. By strategically guiding this process, researchers can discover vulnerabilities more efficiently. To date, fuzzing has successfully identified thousands of bugs in a wide range of software, from general-purpose applications [9] and IoT devices [10] to firmware [11], operating system kernels [12], and database systems [13]. Fuzzing methodologies are broadly categorized as mutation-based, which modifies existing inputs, and generation-based, which creates new inputs from scratch based on a predefined grammar or model.

2. Further details available at:`https://nvd.nist.gov/vuln/detail/CVE-2024-45200`

Algorithm 1: Core Fuzzing Algorithm [14]

```
Input: $initSeedCorpus()$
Output: Expanded corpus and accumulated observations
1  Initialization: ;
2      Initialize the seed corpus ;
3      $queue \leftarrow corpus$ ;
4      $observations \leftarrow \varnothing$ ;
5  while $\neg timeout$ do
6      $candidate \leftarrow choose(queue, observations)$ ;
7      $mutated \leftarrow mutate(candidate, observations)$ ;
8      $observation \leftarrow eval(mutated)$ ;
9      if $isInteresting(observation, observations)$ then
10         $queue \leftarrow queue \cup \{mutated\}$ ;
11         $observations \leftarrow observations \cup \{observation\}$ ;
12     end
13 end
```

Algorithm 1 outlines the fundamental workflow of state-of-the-art fuzzing. The process begins by initializing a queue with a user-supplied set of input, known as the seed corpus. The fuzzer then enters its main loop, which iteratively performs four key operations:

- **Selection:** An input, referred to as a *candidate*, is chosen from the queue.
- **Mutation:** The fuzzer applies modification strategies to the candidate to generate a new, *mutated* input.
- **Evaluation:** The target program is executed with the mutated input, and the resulting behaviour (e.g., code coverage) is captured as an *observation*.
- **Update:** The outcome is evaluated to see if it is interesting. An *observation* is typically considered interesting if it triggers new behaviour, such as exploring a previously unseen code path or causing a crash. If it is, the mutated input that produced it is added to the *queue* for future testing, and the collection of observations is updated.

This feedback loop allows the fuzzer to progressively expand its corpus and explore deeper parts of the target program. The process continues until a stopping condition is met, such as reaching a time limit or achieving a certain level of coverage. The effectiveness of a fuzzer is often measured by the number of unique bugs it discovers and the extent of code coverage it achieves.

1.1 Challenges in Fuzzing Embedded Firmware

Although fuzzing has proven its success in various representations, like source code or binaries, applying it to embedded firmware presents unique challenges. The root of these challenges lies in the inherent characteristics of embedded systems, which often involve tight hardware coupling, real-time constraints, and limited resources, and the limits of current analysis techniques. We organize the challenges into four connected problems that we address in this dissertation.

1.1.1 C1. Fidelity & Throughput

The primary obstacle to firmware analysis is the tight coupling between the software and its specialized hardware. An MCU's firmware is a monolithic binary designed to run on a specific hardware configuration, interacting directly with peripherals via Memory-Mapped I/O (MMIO), Direct Memory Access (DMA), and interrupts [15]. Direct execution on a host is not possible. Re-hosting [16] emulates the system, but accurate peripheral behaviour is hard to model at scale. Abstractions that bypass hardware interactions can boot more targets but miss driver code. Hardware-in-the-Loop (HiL) forwards I/O to real devices [17, 18, 19], which improves fidelity but adds synchronization overhead that slows execution—at odds with high-throughput fuzzing.

1.1.2 C2. Weak Feedback Coverage

Coverage on basic blocks or edges is often too coarse for firmware with heavy use of interrupts, callbacks, and register-level I/O. Conventional coverage may not reflect whether a value actually flows from a definition to its uses in driver code. Fuzzing needs feedback that ties *data flow* to peripheral-facing code to explore deeper behaviour reliably.

1.1.3 C3. Input Validity under Domain Constraints

Many embedded targets accept structured inputs (e.g., PLC ladder logic encodings, field protocols, or language scripts). 90% of testcases generated by mutation produce failed early checks and wasted time [20]. Domain-aware generation or mutation is needed to keep inputs well-formed and to cross protocol and logic checks more often.

1.1.4 C4. From Finding to Preventing Memory Errors

Even with better testing, C-based stacks remain prone to memory bugs. Where possible, architectural support such as Capability Hardware Enhanced RISC Instructions (CHERI) can prevent or trap classes of errors. We need empirical methods to measure such prevention on real embedded software.

These four challenges map to two dimensions in the design space: *where we run* (emulation, HiL, on-device) and *what feedback and inputs we use* (control flow vs. data flow; naive vs. domain-specific). The contributions in this thesis cohere around these dimensions.

1.2 High-Level Research Questions

This section introduces research questions that stem directly from the four challenges discussed in Section 1.1. The questions build upon one another: we first ask how to improve core fuzzing components like input mutation (RQ1) and feedback (RQ2, RQ3), and then use these insights to tackle the ultimate challenges of testing complex targets and evaluating architectural prevention (RQ4).

Research Question 1: *Can domain-specific learning improve mutation so that more inputs pass checks and expose deeper code in PLC workloads?*

Hypothesis: Learning mutation sequences with Sequential Generative Adversarial Networks (SeqGAN) will increase the share of valid and useful test cases for PLC ladder logic, which will raise coverage and bug findings compared with generic havoc-style mutation.

Mutation strategies in fuzzing often leverage bitwise operations to target specific issues, particularly in the American Fuzzy Lop (AFL) [21] through its havoc phase, which randomly applies mutations operators such as bit flips and insertion of significant values. We propose Sizzler, using SeqGAN to optimize these mutation strategies for the havoc phase, thereby enhancing the identification of vulnerabilities in PLC firmware of ladder diagrams through emulation. Sizzler demonstrated its efficiency by swiftly identifying vulnerabilities, securing a CVE-ID, and comparing against traditional fuzzing techniques using the Magma and LAVA-M datasets, thereby proving its broader applicability in embedded systems.

Research Question 2: *Can reconstructed def-use chain coverage provide more useful feedback for fuzzing binaries than traditional edge coverage?*

Hypothesis: Tracking whether values defined at specific sites reach their uses will steer fuzzing to driver and peripheral code that edge coverage alone misses.

Existing research predominantly relies on code or edge coverage derived from control flow graphs (CFGs) for feedback, operating under the assumption that exposing more execution states increases bug detection likelihood. However, the control flow paradigm often provides only a rudimentary approximation of a program's behaviour, a limitation evident in applications where the distinction between control structures and semantic elements is pronounced. To address this, we introduce FuzzRDUCC, which employs dataflow analysis instead of control flow to enhance the granularity of code path coverage. FuzzRDUCC reconstructs the def-use chains through symbolic execution of binaries and implements instrumentation within the emulation process to monitor code coverage. Results indicate that while the def-use chain instrumentation introduces significant runtime overhead, it successfully achieves higher coverage within fixed time budgets.

Research Question 3: *Can on-device fuzzing with hardware breakpoints deliver high-fidelity execution and strong feedback at practical speed for MCUs?*

Hypothesis: Running on the device and using hardware breakpoints to realize def-use feedback will improve throughput and fidelity compared with re-hosting or HiL, while keeping guidance strong.

The feasibility and potential benefits of implementing fuzzing directly on hardware for IoT devices are promising. One key advantage is mitigating the performance overhead typically associated with emulation in binary-only programs. For example, using Quick Emulator (QEMU) is a standard way to emulate the firmware's runtime environment, but the binary-only mode in QEMU introduces a significant tracing overhead, nearly 1300% [20]. To address this, we propose migrating our fuzzing framework, which operates on dataflow, directly onto hardware. This would enable more efficient evaluation of crashes in drivers and peripherals. Our framework uses hardware breakpoints, based on the dataflow graph of the firmwares, to monitor which inputs trigger specific breakpoints. The key idea is to systematically set breakpoints based on the program's dataflow graph and retrieve coverage information by observing which inputs activate these breakpoints. This information can then guide a feedback-driven fuzzing strategy. Since the number of hardware breakpoints within a microcontroller is limited, we strategically place them on a subset of the program's code blocks and periodically relocate them. This approach allows us to balance the limitations of hardware resources while still maintaining effective code coverage and fuzzing performance.

Research Question 4: *As firmware complexity grows, traditional fuzzing struggles with highly structured inputs like language interpreters. How can we evolve test generation beyond simple mutation to rigorously assess architectural defences, thereby measuring the shift from vulnerability discovery to prevention?*

Hypothesis: For complex, stateful targets like the MicroPython interpreter, naive fuzzing is ineffective. A combination of LLM-seeded test generation and concrete-syntax-aware (CST) mutation is required to create valid inputs that can penetrate deep logic and expose memory-unsafe states. Applying this advanced testing methodology within a differential framework will empirically demonstrate that an architectural solution like CHERI prevents entire classes of memory errors that a standard build would suffer from.

We evaluated CHERI's effectiveness in preventing memory errors using differential testing on MicroPython, a widely-used embedded interpreter chosen for its complex codebase and large attack surface. Our method involves running an identical suite of tests on both a CHERI-enabled build and a standard, non-CHERI build.

To generate relevant and diverse testcases, we seeded a large language model (LLM) with public bug narratives. We then applied concrete-syntax-tree mutations to these inputs to ensure they remained well-formed and syntactically correct. By comparing the execution logs and crash traces from both builds, we could precisely identify memory errors that CHERI successfully trapped, which would otherwise cause crashes or silent data corruption in the standard build.

1.3 Thesis Statement

The security of microcontroller firmware can be significantly advanced by adopting a hardware-centric paradigm that not only improves vulnerability detection by replacing slow emulation with direct on-chip fuzzing guided by intelligent, data-flow-driven analysis, but also pioneers vulnerability prevention through architectural memory safety enhancements.

Figure 1.1 ties the thesis to the end-to-end fuzzing loop. At the top, the **fuzzing engine** reads a seed from the corpus, applies a **mutation strategy**, and sends the final input to the **Program Under Test (PUT) executor**. The **green path** shows a run that raises coverage (first input `0x01` reaches new code across State 1 and State 2). The **red path** shows a run that triggers a bug (second input `0x11` reaches a faulting path). The **bug monitor** collects signals (coverage deltas, exits, timeouts, crashes), verifies crashes, and feeds results back to the engine—closing the loop of Algorithm 1.

The bottom row links each contribution to the correct module:

- **MicroPython (left, green) —Seed set.** This pipeline builds the *seed corpus*: it aggregates CVE proofs of concept (PoC) and issue reports, uses an LLM-based suspicious-input generator, and applies a CST-preserving mutator to keep inputs valid. We also run the same inputs on non-CHERI and CHERI builds to label seeds that expose memory issues; the primary output is a high-quality, unified seed set for the fuzzer. (**RQ4**, mainly C3; CHERI results inform C4.)
- **Sizzler (centre, blue) —Mutation strategy.** A SeqGAN learns *sequences of mutation operations* that pass PLC checks and open new paths in ladder-logic programs under emulation. Sizzler plugs into the mutation stage of the engine and improves how seeds are turned into final inputs. (**RQ1**, C3.)
- **Hardfuzz (right, gray) —Executor.** The executor runs *on the device.* On the host we keep the mutation engine, coverage map, and a small state machine; we load a *def–use chain* and use JTAG to arm hardware breakpoints so the device itself provides fast, high-fidelity execution and feedback. (**RQ2, RQ3**, C1–C2.)
- **FuzzRDUCC (right, gray, inside Hardfuzz)** It is the *feedback layer* that adds *def–use* coverage during emulated execution: QEMU instrumentation records def and use events and updates the coverage map to guide seed selection (**RQ2**, C2). In Hardfuzz, the same idea is realized with *hardware breakpoints* instead of QEMU hooks.

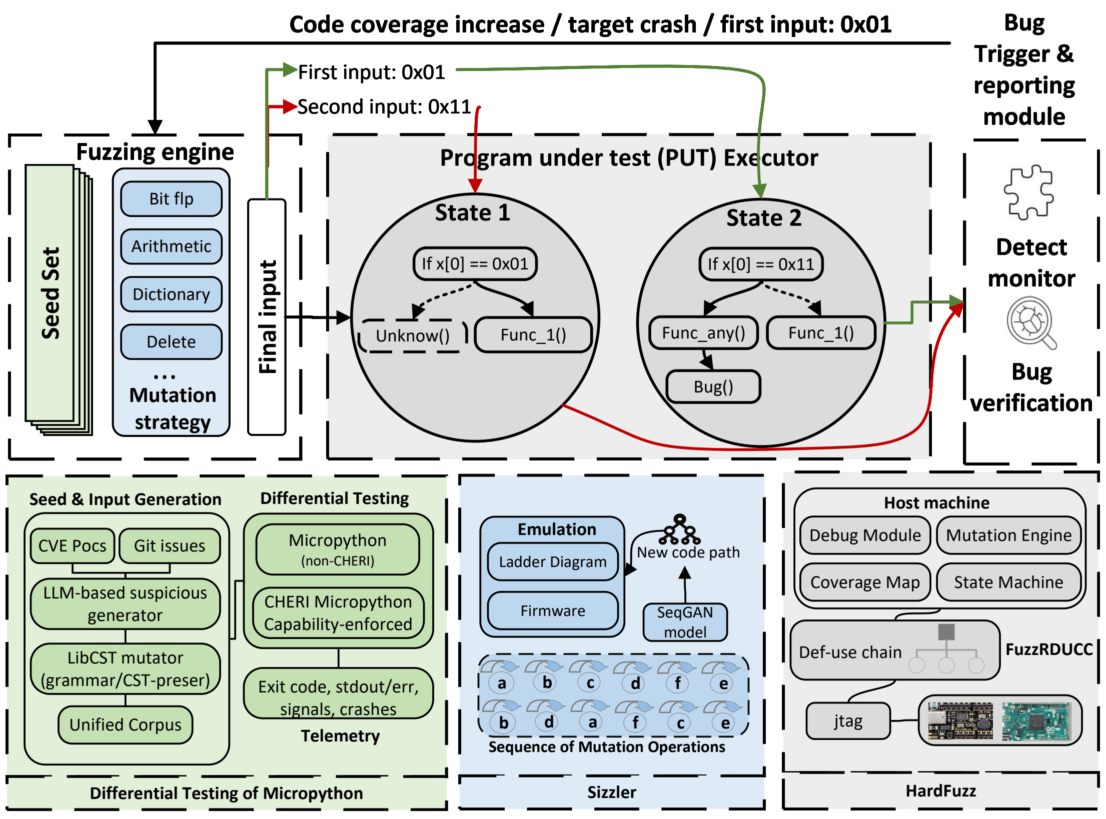


Figure 1.1: End-to-end fuzzing stack and placement of contributions. Top band: seed → mutation → executor; Bottom (modules): Differential Testing for MicroPython (left), Sizzler (centre), Hardfuzz (right)

1.3.1 Sizzler

To overcome the wasting of time caused by the randomness of mutation strategies in fuzzing process, most existing approaches rely on imprecise heuristics or complex and expensive program analysis (e.g., symbolic execution or taint analysis) techniques to generate and/or mutate inputs to bypass the sanity checks. For example, MOPT [22] uses a set of heuristics to optimize the mutation strategies in the havoc phase of AFL. However, these heuristics are not adaptive and may not work well for all programs. HavocMAB [23] uses a multi-armed bandit algorithm to learn the best mutation strategies for each program. However, this approach requires many iterations to converge to a good strategy and may not work well for programs with complex input formats.

To address this limitation, we propose Sizzler (Sequential Fuzzing in Ladder Diagrams for Vulnerability Detection and Discovery in Programmable Logic Controllers), shown in figure 1.1, a novel approach that employs Sequential Generative Adversarial Networks (SeqGAN) to optimize mutation strategies for the havoc phase of AFL, thereby enhancing the identification of vulnerabilities in PLC firmware of ladder diagrams through emulation and HiL. This work is related to mutation strategy in fuzzing process, shown in blue area. SeqGAN is utilized to learn the logic of mutation operations within the executed PLC code, thereby aiding the fuzzing process [24]. The use of SeqGAN increases the number of test cases that are likely associated with potential PLC code vulnerabilities and enhances the rate of code path discovery.

Sizzler's performance is assessed using a practical, vendor-independent emulation test bed constructed with the OpenPLC [25] framework. In this environment, converted ladder diagrams are executed as binaries on de facto MCUs, eliminating the need for re-hosting PLC firmware. This approach promotes a realistic method for accelerating the study of PLC vulnerability discovery without dependencies on vendor-proprietary PLC code.

Moreover, the benefits of Sizzler's mutation strategy are demonstrated beyond PLCs and within the wider embedded systems context. This is achieved through comparisons with other fuzzing strategies using the LAVA-M [26] and Magma [27] datasets as benchmarks. Our results indicate that Sizzler outperforms the majority of state-of-the-art fuzzing schemes.

By leveraging the capabilities of SeqGAN, Sizzler enhances the efficiency and effectiveness of fuzzing by generating more targeted and diverse test cases. This approach not only improves the detection of vulnerabilities in PLC systems but also offers broader applications in embedded systems security.

1.3.2 FuzzRDUCC

The core of improvement of fuzzing is determining the fuzzing's direction, as well as how and where to mutate the input. Most existing fuzzers rely on code coverage or edge coverage derived from control flow graphs (CFGs) as feedback to guide the fuzzing process, operating under the assumption that exposing more execution states increases the likelihood of finding bugs. However, this control flow paradigm often provides only a rudimentary approximation of a program's behaviour, a limitation that becomes particularly evident in applications where the distinction between control structures and semantic elements is pronounced. For instance, in firmware with extensive use of interrupts, callbacks, and register-level I/O, traditional coverage metrics may not accurately reflect whether a value defined at a specific site actually reaches its uses in driver code. Edge coverage does not show whether critical values reach their uses in driver code. Firmware heavy with interrupts and MMIO benefits from feedback that reflects *dataflow*.

Hence, we propose FuzzRDUCC (Fuzzing with Reconstructed Def-Use Chain Coverage) to leverage def-use chains for capturing the dataflow of the target, providing feedback instead of relying solely on control flow. FuzzRDUCC is designed to enhance the capabilities of fuzzers through the incorporation of def-use chains, structuring our approach into two distinct phases: static analysis and fuzzing.

In the static analysis phase, we employ the angr [28] framework to extract def-use chains from binary code. This involves acquiring precise addresses and the quantity of definitions and usages for each block translated. The process includes the instrumentation of def-use chains, which entails recording the number and address of definitions and usages for every translated block, facilitated through the lightweight code generation capabilities of QEMU.

Transitioning to the fuzzing phase, our methodology integrates the deployment of an innovative bitmap specifically designed to precisely monitor the locations of definitions and usages. This is achieved through instrumentation based on the addresses of definitions and usages within basic blocks translated from QEMU. Upon execution of a basic block, the bitmap is updated in comparison with a global map to track the execution state. This mechanism acts as a directive for the fuzzer, guiding the initiation of a re-mutation process informed by the analysis of previously evaluated seeds.

1.3.3 Hardfuzz

State-of-the-art firmware fuzzing uses *rehosting*, *para-rehosting*, *hardware-in-the-loop* (HiL), and *fully on-device* execution, but each approach has practical limits. *Rehosting* runs firmware in a virtualized target, yet setup is complex and often slow. *Para-rehosting* lowers emulation costs via HAL stubs, but does not reach driver code. *HiL* forwards I/O to real devices, which preserves fidelity but adds heavy synchronization overhead. *Fully on-device* tracing (e.g., Intel PT, ARM ETM) is not widely available across microcontrollers and boards.

We present Hardfuzz, a hardware-first fuzzing that uses *hardware breakpoints* as its feedback channel. We statically extract definition–use (def-use) chains and place a hardware breakpoint at each *def*. When the def fires, we step off the site and re-arm breakpoints at the corresponding *uses*, which lets us follow value flows at instruction granularity. The fuzzer records both *def hits* and *def-use pairs* in two coverage maps to guide input selection. Because microcontrollers expose only a few Flash Patch and Breakpoint (FPB) slots, we employ a *weighted relocation* policy: we prioritize defs with more uses and gradually downweight defs that have already been explored; use breakpoints are inserted as temporary hardware BPs, so a slot frees itself on hit. This design gives precise, low-overhead guidance while staying fully on hardware.

By combining on-device execution with def-use-driven feedback, Hardfuzz improves path discovery in driver and embedded code while avoiding rehosting complexity and HiL synchronization costs. The framework is practical for IoT targets with limited debug resources and scales with automatic corpus growth and mutation.

1.3.4 Differential testing of MicroPython under CHERI

Embedded stacks often include high-level interpreters. Our early fuzzing showed that naive mutation yields many invalid scripts; more importantly, memory-unsafe C interpreters remain vulnerable even when fuzzed. We need both better input generation and architectural support to prevent memory errors.

We study *MicroPython*, a Python 3 implementation for MCUs. We build a *differential testing* framework that runs the same tests on a standard build and a CHERI-enabled build. The framework uses a Large Language Model with public MicroPython CVE narratives and bug reports to generate valid starter tests for seed collection, and then apply syntax-aware mutations on the CST. We compare logs and crash dumps: a crash in the non-CHERI build that is trapped as a bounds violation in the CHERI build points to a memory safety issue that CHERI prevents.

The differential framework helps us find more unique core-interpreter memory-safety bugs (excluding libffi), while total unique bugs are higher on non-CHERI due to libffi presence. It also helps us find 35 unique bugs on the latest version of MicroPython (MicroPython-1.27-preview). We also create a dataset of MicroPython memory safety bugs to support CHERI and embedded security research.

1.4 Contributions

The thesis makes the following original contributions to the field of embedded firmware security, particularly in the context of fuzz testing for embedded system. Each contribution addresses specific challenges and research questions outlined earlier.

1. **Domain-aware mutation for PLC workloads (Sizzler).** A SeqGAN-based mutation strategy that increases valid test rate and improves coverage on PLC ladder logic. We also extend emulation with GPIO/I2C and Modbus/TCP refinements and release a PLC ladder logic program dataset for future security research. *(RQ1, C3).*
2. **Dataflow-guided fuzzing with def-use coverage (FuzzRDUCC).** A binary-focused method to reconstruct def-use chains and a lightweight QEMU instrumentation that tracks def-use events at runtime to guide fuzzing beyond edge coverage. *(RQ2, C2)*
3. **On-device fuzzing with breakpoint feedback (Hardfuzz).** A practical on-hardware fuzzing framework that realizes def-use guidance via hardware breakpoints with a weighted relocation policy under tight hardware breakpoint limits, avoiding re-hosting and HiL overheads. *(RQ3, C1, C2)*
4. **Differential testing of MicroPython under CHERI.** A differential testing framework that combines LLM-seeded, CST-aware test generation with CHERI-based and non-CHERI execution to expose and prevent memory errors in MicroPython. We also present a curated dataset of MicroPython memory safety bugs to support CHERI and embedded security research. *(RQ4, C3, C4)*

1.5 Publications

This thesis draws on the following publications and preprints. Each item notes the related research question(s) and challenge(s).

- **Sizzler: Sequential Fuzzing in Ladder Diagrams for Vulnerability Detection and Discovery in Programmable Logic Controllers** Addresses **RQ1** (C3). *Publication:* [IEEE Transactions on Information Forensics and Security, DOI: 10.1109/TIFS.2023.3340615].

- **FuzzRDUCC: Fuzzing with Reconstructed Def-Use Chain Coverage** Addresses **RQ2** (C2). *Publication:* [Doi:https://doi.org/10.48550/arXiv.2509.04967].
- **Hardfuzz: On-Device Def-Use-Guided Fuzzing with Hardware Breakpoints.** Addresses **RQ3** (C1, C2). *Under Review: The 5th International Fuzzing Workshop (FUZZING) 2026.*
- **Differential Testing of MicroPython under CHERI.** Addresses **RQ4** (C3, C4). *Under Review: 2026 European Conference on Object-Oriented Programming (ECOOP 2026)* .

1.6 Summary of Research Artifacts

To facilitate reproducibility and support further research, this thesis is accompanied by a comprehensive set of research artifacts. These artifacts encompass the source code for the proposed techniques, experimental datasets, and records of upstream bug reports and patches. All materials are publicly accessible via the following repositories:

- **Sizzler implementation and PLC corpus.** The source code for the Sizzler framework, comprising the ladder-logic mutation engine, the PLC emulation harness, and the automation scripts required to reproduce the experiments in Chapter 3. This repository also contains the synthetic ladder logic programs and the vulnerability corpus used for evaluation. `https://github.com/MaksimFeng/Sizzler`
- **FuzzRDUCC prototype.** The implementation of the FuzzRDUCC data-flow-guided fuzzer (Chapter 4). This artifact includes the Angr-based scripts for definition-use chain reconstruction, the fuzzing workflow, and the configuration files for the `binutils` targets. `https://github.com/MaksimFeng/qemuafl-dataflow`

- **Hardfuzz prototype.** The complete source code for Hardfuzz (Chapter 5), including the on-device breakpoint controller, the definition-use selection policy, and the patched GDBFuzz baseline used for comparison. The repository also includes the firmware images and harness code for the three MCU targets. `https://github.com/MaksimFeng/Hardfuzz`
- **MicroPython differential testing framework.** The testing framework presented in Chapter 6, which includes the test generation pipeline (leveraging LLM prompts and CST-based mutators), the harnesses for executing MicroPython on CHERI and non-CHERI targets, and the scripts used to collect, classify, and de-duplicate differential outcomes. `https://github.com/MaksimFeng/ML4Secure`
- **Bug reports, CVEs, and patches.** A comprehensive record of confirmed bugs, and patches resulting from this work (including detailed MicroPython bug reports) is provided in Appendix B.

1.7 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 1 This chapter introduces the core problem we are tackling. We outline the main challenges (C1–C4), state our high-level research questions (RQ1–RQ4) that guide this work, and present the thesis statement and an overview of the following chapters.

Chapter 2: Background and Related Work This chapter covers the essential background information needed to understand this thesis. We explain the basics of MCUs and PLCs, review different software testing approaches, and introduce key concepts like automated testing, data-flow analysis, and the CHERI secure hardware architecture.

Chapter 3: Sizzler (RQ1, C3) This chapter presents Sizzler, a new tool we developed to find bugs in PLCs. We describe how Sizzler automatically creates tests specifically designed for ladder logic with new mutation strategy, the programming language used by PLCs. This work directly answers our first research question (RQ1).

Chapter 4: FuzzRDUCC (RQ2, C2) In this chapter, we introduce FuzzRDUCC, a technique for making automated testing more effective by tracking how data moves through a program. We explain how we analyse a program's code to understand these data flows and use that information to guide our bug-finding efforts. This chapter addresses our second research question (RQ2).

Chapter 5: Hardfuzz (RQ3, C1–C2) This chapter details Hardfuzz, our approach for testing software directly on the physical device itself. We describe our technique for tracking data flow in real-time on the hardware, which is difficult due to its limited resources. We then show how this on-device method compares to testing in a simulated environment, addressing our third research question (RQ3).

Chapter 6: Differential Testing of MicroPython under CHERI (RQ4, C3–C4) This chapter explores the security benefits of modern CHERI hardware. We present a differential framework that uses LLM and libCST to generate testcases to test the MicroPython programming language running on CHERI and non-CHERI. The goal is to measure how well this special hardware prevents common and critical memory bugs, answering our fourth research question (RQ4).

Chapter 7: Conclusion Summarizes contributions, limitations, and future work.

Chapter 2

Embedded Fuzzing: Challenges and State of the Art

Embedded systems (such as IoT devices, industrial controllers, medical implants, etc.) present unique challenges for software testing. Their firmware is tightly coupled to specialized hardware and peripherals, and they often run on bare metal or a small RTOS, and targets diverse architectures. Fuzz testing–repeatedly executing a program with mutated inputs to trigger faults – has proven highly effective at exposing bugs in conventional software, and is recommended by multiple industry standards.

However, applying fuzzing to embedded firmware is non-trivial. Unlike user-space programs that can be instrumented and run in a process on a PC, firmware is designed to run on a specific microcontroller with particular memory-mapped I/O and device drivers. Simply compiling firmware code as a normal application or fuzzing it in isolation fails to exercise interactions with the actual hardware. The holistic fuzzing of embedded systems must cover the firmware and its hardware context. Two fundamental issues make this difficult: (1) the strong dependence on specific hardware, and (2) the immense heterogeneity of architectures and peripherals in the embedded world. These factors lead to a lack of a ”one-size-fits-all” embedded fuzzing solution. Recent surveys confirm that fuzzing embedded systems remains an open research problem and no single golden solution exists yet [29].

2.1 Core Fuzzing Components for Embedded Systems

No matter the execution environment (real hardware, emulator, or hybrid), an embedded fuzzer consists of several fundamental components:

1. **Seed Selection**: How to select initial inputs.
2. **Mutation Strategy**: How it mutates inputs and schedules test cases
3. **Feedback Scheme**: What feedback metrics guide it (coverage or other fitness functions).
4. **Bug Detection**: How it detects and handles faults.

In this section, we discuss how these components are realized or adapted in state-of-the-art embedded fuzzing systems.

Seed generation and input corpus: Like any fuzzing campaign, an embedded fuzzing effort starts with an initial set of test inputs (the seed corpus). For firmware, what constitutes an ”input” can vary widely. It could be a sequence of bytes sent over a communication interface, sensor readings over time, a file loaded from flash memory, or even a sequence of UI actions. Selecting good seeds is essential to bootstrap coverage. In some cases, researchers use recorded real-world inputs, e.g., network traces or sensor logs, as seeds to ensure the fuzzer begins in a valid state. If the target firmware has an associated specification (e.g., a network protocol or file format), seeds may be constructed from known valid examples in that format [30]. Some works have applied grammar-based fuzzing to embedded inputs: for instance, if fuzzing a smart light’s wireless protocol, one can supply a basic valid packet as a seed, then let the fuzzer mutate its fields [31]. Another strategy, used in P2IM [6] and others, is to start with a dummy seed (like an empty or random input) and rely on the firmware’s own initialization to generate a starting state; the fuzzer then begins mutating whatever input bytes the firmware consumed. In scenarios where the firmware expects a complex sequence (e.g., a command handshake), the harness often provides a fixed prologue to set up the state, and fuzzing is applied only to the variable part of the input [32]. This effectively means the seed includes the fixed script of actions. An example is fuzzing a device that first requires login: the harness can always send a correct login sequence (not fuzzed) and then fuzz the subsequent payload. As the corpus evolves, embedded fuzzers also employ seed minimization and interesting test case selection akin to AFL. That is, when new coverage is found, the input that caused it is added to the corpus. Some frameworks, like the one by Zhao et al., specifically consider how to reduce the size of the input space by splitting firmware into independent components and fuzzing them separately [33], by doing so, they effectively generate smaller ”modular” seed corpora for each component, rather than one huge corpus for the entire firmware. In general, seed generation for embedded fuzzers often requires more manual setup than for, say, fuzzing a file parser on a PC. The harness or test-driver code must feed the input into the firmware in the correct manner-whether through writing to a memory buffer (in emulation), sending over a serial port, or toggling a General-Purpose Input/Output (GPIO) pin in hardware. If this is done incorrectly, the firmware might not accept the input at all. Thus, a thorough understanding of the firmware's expected inputs (via documentation or reverse engineering) greatly aids the creation of an effective initial seed set [34].

Mutation strategies and scheduling: Once a corpus of inputs is established, an embedded fuzzer mutates them to generate new tests. Most frameworks simply reuse classic byte-level mutation operators from tools like AFL [21]: random bit flips, increments/-decrements, inserting or deleting bytes, swapping chunks, etc. These remain effective for low-level firmware data (which often lacks complex structure like deeply nested formats). A few frameworks introduce domain-specific mutations. For example, if fuzzing a sensor input that is a 16-bit analogue reading, one might mutate it with biases towards boundary values (0x0000, 0xFFFF) that could trigger edge conditions in calibration code [35].

Another example is multi-stage mutation: Yu et al. suggested a multi-stage generation for IoT protocols, where initial mutations ensure the message remains parseable, and later mutations target deeper fields. In practice, many embedded fuzzers still use dumb mutations, relying on coverage feedback to eventually favour those that lead to new states [36].

Scheduling refers to which input from the corpus is chosen next to fuzz and how long to fuzz it (the energy given to an input). AFL's default power schedule (favouring smaller, recently fruitful seeds) is often adopted [37]. One peculiarity in embedded fuzzing is the presence of long-running stateful sequences. If an input is actually a sequence of operations (e.g., a series of CAN bus messages), the fuzzer might need to mutate the whole sequence or parts of it. Some frameworks explicitly maintain stateful sequences and try mutating one message at a time while keeping the others fixed, which is akin to higher-level scheduling of sub-inputs [38]. The concept of ”fragmentation” of inputs for scheduling was explored by Amini et al. in the context of protocol fuzzing (Sulley framework)-modern embedded fuzzers implicitly use similar ideas when they allow, say, one sensor input to vary while others remain constant for a while, to isolate the effect on coverage [39].

An important aspect of scheduling in hardware fuzzing related to timeouts and resets. Since firmware may enter a hung state (e.g., waiting forever for a sensor), fuzzers must detect that and reset the environment to avoid stalling. Many tools implement a global timeout for each test case. For example, GDBFuzz [40] will reset the device if an input does not complete execution within a certain time window (which is set empirically for each specific target). Scheduling also includes deciding when to reset internal state: some fuzzers reset after every input, others allow a series of inputs to be given in one session if the protocol is interactive [20]. For instance, to fuzz a device that processes a continuous stream, RFUZZ proposed by Laeufer [41] feeds a dozen mutated inputs in sequence before a reset, to simulate ongoing operation-essentially treating each sequence as one compound test case.

Coverage and fitness metrics: Coverage-guided fuzzing dominates the landscape of embedded fuzzing approaches, as evidenced by the fact that nearly all recent works integrate some coverage measurement [42]. The standard metric is code coverage-usually edge coverage or basic-block coverage similar to AFL's notion. In a hardware fuzzer, this might be approximated (e.g., GDBFuzz's partial coverage via breakpoints is a coarse-grained block coverage). In an emulator, it's straightforward to instrument every basic block or jump. AFL-style edge coverage (with a global bitmap of edges hit) has been implemented in many firmware fuzzers that run in QEMU or Unicorn engine [43, 44]. Some works consider additional metrics: for example, if focusing on a vulnerability like memory corruption, one might treat a detected invalid memory access as a special feedback (not just a crash but a ”red flag” to be reached) [45]. Avatar-based fuzzers introduced the idea of monitoring for silent memory corruptions in emulated execution-by comparing certain memory regions between the emulator and hardware to see if corruption occurred without an immediate crash [46]. This acts as a fitness signal to guide the fuzzer toward inputs that cause memory inconsistencies (which often indicate latent corruption). Another metric sometimes used is path length or execution depth, to reward inputs that drive the firmware further (especially useful if the firmware has a long init phase-you want inputs that survive longer) [47]. Agamotto presented by Wang et al. applied a time-to-execute metric to guide firmware fuzzing under resource constraints [48].

In general, however, branch coverage remains the primary fitness metric. The surveys note that while code coverage is an imperfect proxy for bug-finding, it is the easiest and most generic measure to implement and has correlated well with finding crashes in practice. Klees and Schloegel. separately argued for using the number of bugs found as the ultimate metric, but since ground-truth bugs are unavailable for most firmware, coverage is used as a surrogate [14, 49].

Crash detection and handling: Determining when a fuzz test triggers a fault in embedded context can be tricky. In native fuzzing, a crash usually manifests as a process exception (segfault, etc.) [50]. In firmware, there is no process isolation-a fault may simply reset the device or set an error flag. On real hardware, fuzzers commonly detect a crash by monitoring the debug interface or a heartbeat [51]. For example, if the device enters a fault handler (many ARM Cortex-M MCUs have a usage fault or hardfault handler for exceptions) [52], the fuzzer can detect that via the debugger (e.g., a breakpoint on the fault handler). Alternatively, a simple liveness check is used: if the device stops responding (no longer hits the breakpoint or produces output) for a certain time, it's assumed to have crashed and is reset. In emulation, crashes can be caught by the emulator (illegal memory access in QEMU, etc.). Many fuzzers also inject assertions or canaries to detect memory corruption that doesn't immediately crash. For instance, some works utilize all kinds of sanitizers in the emulator to catch out-of-bounds [53, 54, 55], or watchpoints on key memory regions. The GDBFuzz authors mention the concept of detecting silent memory corruptions on devices without MMUs by instrumenting an emulator to watch memory writes. Once a crash is detected, the fuzzer will record the input that caused it, possibly minimize it, and then continue. Uniqueness of crashes is often determined by the fault address or signature (as in AFL). One must be careful on hardware because repeated occurrences of the same bug might manifest slightly differently if nondeterminism is involved (e.g., race conditions causing crashes at varying addresses) [56].

Performance optimizations: While not a "component" per se, it's worth noting how embedded fuzzers optimize performance. On hardware, concurrency is limited, but one can still parallelize by using multiple devices. Some projects used an array of development boards to fuzz in parallel, each assigned different seeds [57]. On multi-core MCUs or SoCs, one could fuzz multiple instances of a firmware component if isolated (though this is uncommon). In emulation, parallel fuzzing is easier [58]-one can run many emulator instances on a PC cluster, just as with normal fuzzing. The bottleneck is often the emulator speed or the constraint solver speed (for hybrid approaches). Techniques like snapshot/restore are employed: e.g., Snappy introduced a fast snapshot mechanism for QEMU to quickly reset firmware state without reloading from scratch [59]. This dramatically increases test throughput by avoiding full re-initialization of the firmware each time.

Finally, to round out the discussion, evaluation methodology. Klees et al.'s work showed that fuzzer evaluations can be misleading if not properly controlled [14]. Eisele et al. [40] echo that the embedded fuzzing field would benefit from standard benchmarks and performance metrics to compare approaches. They suggest adapting baseline fuzzing evaluation principles to firmware: using a diverse suite of firmware programs with known bugs, measuring bugs found over time, not just coverage. There is not a universally adopted benchmark like LAVA or DARPA CGC for the embedded domain, though some initial collections exist (RIOT os test suite, Juliet test cases ported to embedded, etc.). Researchers are aware of this gap and are moving towards more rigorous comparisons. Another bottleneck is throughput. Emulators exploit snapshot/restore to skip init phases (fast reboot) and run many instances in parallel [43]. On hardware, frameworks avoid reflashing by looping a harness in place and only power-cycling on hangs. Hybrid systems proxy only hard peripherals to real hardware to keep most execution local [19]. Trace-based feedback reduces in-target work and keeps runs near native speed [20].

In conclusion, the core components of embedded fuzzers are fundamentally similar to those of traditional fuzzers, but their implementation must be tailored to the constraints of firmware and devices. Seed selection must account for firmware's context, mutation and scheduling must often deal with stateful sequences and slow resets, coverage collection might require innovative use of hardware features, and crash detection can be non-standard.

2.2 Hardware-Based Fuzzing on Real Devices

Over the last few years, a plethora of tools and techniques have emerged to enable fuzz testing of firmware under various conditions [60]. Researchers have explored approaches ranging from running the firmware on real hardware with instrumentation, to fully emulating the hardware in software, to hybrid combinations in between [16]. Each approach must balance fidelity (how accurately the execution matches a real device) against automation and speed (how much manual effort or slowdown is incurred), shown in the figure 2.1.

As illustrated in the figure 2.1, running firmware on an actual device yields perfect hardware fidelity (GDBFuzz), but instrumenting or controlling the execution can be complex and slow. In contrast, pure software emulation allows easier introspection and faster resets, but may suffer from incomplete device models or incorrect peripheral behaviour [40]. In practice, existing tools occupy different points in this design space, and combining their advantages is an active area of research. We examine all major approaches, from hardware-in-the-loop setups to full firmware emulation, and discuss their core components (how they generate inputs, mutate and schedule tests, monitor coverage, and determine fitness or crashes).

One straightforward way to fuzz firmware is to run it on the actual hardware and treat the device as the System Under Test (SUT). This avoids the difficult problem of modelling the device's behaviour in software-i.e. it achieves the highest fidelity by definition. The challenge then is how to provide feedback (coverage, fault detection) from the device and how to drive it with test inputs at scale [62]. Standard coverage-guided fuzzers like AFL assume they can instrument the target program to collect coverage metrics and reset it quickly between test cases. In an embedded context, the firmware cannot usually be

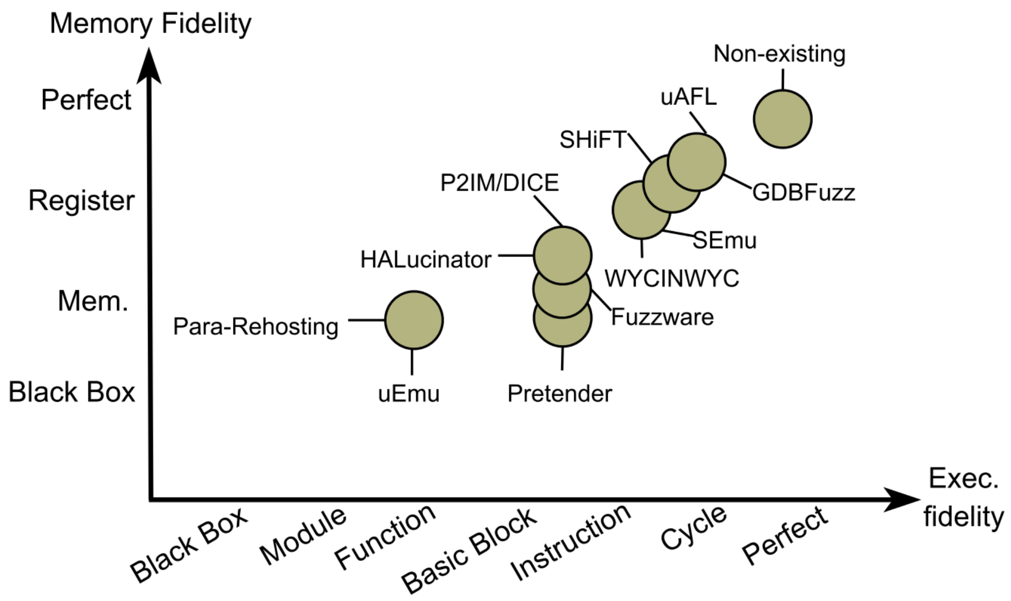


Figure 2.1: The trade-off between fidelity and automation/speed in embedded fuzzing approaches. [61]

instrumented or even paused without special support. Moreover, embedded boards often lack an OS to assist in error handling or I/O, crashing the firmware may simply hang the device. Recent work has therefore leveraged hardware features and external debug interfaces to enable on-device fuzzing of firmware [63, 64].

Hardware instrumentation (via debug interfaces): Many microcontrollers include a debug port (e.g. ARM CoreSight with SWD/JTAG, or similar on other architectures) that allows an external debugger to control execution. Tools like **µAFL** and **GDBFuzz** (Eisele et al.,) [40] take advantage of this to perform coverage-guided fuzzing on physical boards. The basic idea is to use hardware breakpoints to detect when new code is reached during execution, without needing to instrument the binary. For instance, GDBFuzz configures a limited set of breakpoints at strategic code locations and runs the firmware until a breakpoint hits; each unique hit corresponds to a new coverage point. By dynamically managing breakpoints (swapping them in and out) and iteratively feeding inputs via a harness, these frameworks gather coarse-grained coverage feedback from the device. Eisele et al. report that GDBFuzz could find known and new bugs on several ARM Cortex-M boards, achieving substantially higher code coverage than black-box testing despite no firmware instrumentation and only minimal slowdown from the debugger interface [40]. Similarly, AFL uses the on-chip debug module to single-step or break on branches, assembling coverage feedback in a ”non-intrusive” way so that even closed-source firmware can be fuzzed without modifications [65]. These on-device approaches essentially treat the microcontroller as the execution engine while using a connected PC to run the fuzzer logic. They must cope with the typically limited throughput of hardware fuzzing-resetting or flashing a device and waiting for it to run is far slower than in-memory execution. To mitigate this, hardware fuzzers often keep the device running in a loop to avoid reboots, and only restart it when a crash or hang is detected. They may also restrict coverage collection to certain code regions (due to the small number of hardware breakpoints available). Despite these constraints, the use of real hardware ensures that all peripherals and timing conditions are accurate, avoiding false positives that might arise in emulation. It has been demonstrated that such hardware-in-the-loop fuzzers can be ”versatile” and require surprisingly little target-specific customization-any board that GDB can attach to could, in principle, be fuzzed this way. The downside is scalability: a separate physical device (or at least a separate debug probe) is needed for each parallel fuzzing instance, and the execution speed is bounded by the device’s performance and I/O latency to the host.

Hardware trace-assisted fuzzing: In cases where setting breakpoints is too intrusive or limited, another approach uses trace hardware to obtain coverage. Modern processors often have features like Intel PT (Processor Trace) or ARM ETM (Embedded Trace Macrocell) that can stream out information about executed branches with minimal overhead [66]. Although primarily used on high-end systems, researchers have applied these to fuzzing as well [20]. For example, Nagy et al. showed that Intel PT could be leveraged to significantly reduce the overhead of coverage collection for fuzzing, essentially running the target at near native speed while logging coverage externally. In an embedded context, if a microcontroller provides a trace port, a fuzzer could use it to know which basic blocks or branches were executed by each input-effectively an off-chip coverage oracle. One can view this as a special case of hardware-in-the-loop fuzzing where the feedback channel is a high-bandwidth trace stream instead of breakpoints. For instance, POTUS (an academic prototype for ARM Cortex-M) used the ETM trace unit to capture execution profiles of firmware under test, enabling on-device coverage-guided fuzzing without modifying the code (the trace data is parsed on the host to compute coverage) [64]. These trace-based methods achieve full instruction coverage fidelity with low perturbation of the timing, but require that the target chip have a supported trace interface and that the fuzzer can process the trace data fast enough. As trace ports are not present or accessible on all devices (and sometimes disabled for security on production units), this approach is powerful but not universally applicable [67].

Side-channel feedback: An intriguing variant of hardware-based fuzzing is to infer program coverage or state by observing physical side channels (such as power consumption or electromagnetic emanations) rather than digital outputs. Sperl and Böttinger introduced a side-channel-aware fuzzing technique in which an oscilloscope monitors the device' s power usage to guess which code paths were executed. By correlating segments of the power trace with known ”fingerprints” of basic blocks or functions (obtained through offline training), the fuzzer can estimate when a new region of code has been hit. This provides a feedback signal analogous to coverage, even though the firmware is running uninstrumented on a real device. In their case study on an 8-bit microcontroller, the side-channel fuzzing approach successfully guided the fuzzer to cover more code and find vulnerabilities, with the advantage that it needed no special hardware beyond the power sensor [68]. The limitation is that distinguishing execution paths via side-channel measurements can be error-prone and device-specific-noise or slight program changes can confuse the classification. Nonetheless, this idea expands the toolbox for scenarios where standard debugging interfaces are unavailable, but physical monitoring is possible (e.g., fuzzing a sealed device by measuring its power draw or timing). Another form of side-channel feedback could be timing analysis: for example, if a particular input causes a processing loop to run longer (observed via a simple timing measurement), that might indicate new code was exercised [69]. While cruder than code coverage, such timing-based feedback can still guide fuzzing of systems where fine-grained instrumentation is infeasible.

Direct protocol interface fuzzing: Not all embedded fuzzing needs internal coverage feedback. In practice, many vulnerabilities in embedded devices (especially networked IoT devices) have been found by classic black-box fuzzing of their communication interfaces-treating the device as a remote server, for instance, and sending malformed network packets or peripheral inputs [70]. This approach reuses the device's existing message interfaces (UART consoles, network sockets, USB endpoints, etc.) to inject test cases, and monitors for crashes by detecting when the device resets or becomes unresponsive. It requires minimal setup: essentially the fuzzer acts like a rogue client to the device. Tools like Boofuzz (an open-source network fuzzer) have been widely used in this manner to fuzz IoT firmware over protocols like HTTP, Bluetooth, or proprietary command interfaces [71]. Academic works have augmented this with some domain knowledge; for example, FirmFuzz (Srivastava et al.,) performs introspection on firmware images to identify potential input vectors (e.g., network message handlers or file parsers in the firmware) and then generates inputs for those interfaces [72]. The major drawback of black-box interface fuzzing is the lack of feedback: without coverage information, it may spend a long time exploring ineffective inputs. Additionally, many embedded protocols require valid stateful sequences (e.g., an authentication handshake before sending the payload), so the fuzzer must be aware of protocol semantics or have recorded traffic to replay. Recent research on stateful fuzzing and targeted fuzzing can help in this regard. For instance, Natella et al. proposed StateAFL, a greybox fuzzer that is aware of protocol states and can reset the target to known good states between tests [73]. In embedded scenarios, one might implement a similar strategy: reset the device or reinvoke a subsystem to a clean state for each test, possibly via hardware watchdogs or external power cycling if necessary.

In summary, hardware-based embedded fuzzing provides the highest realism-the firmware is exercised on its real platform-and recent advances show that even coverage-guidance is possible through clever use of debugging and side-channel techniques. These methods shine in finding bugs that depend on actual hardware behaviour (timing, concurrency, precise register states) that emulators might miss. However, they face significant scalability and automation challenges: setting up and controlling physical devices is labour-intensive. This motivates the complementary line of research: re-hosting firmware in an emulated or simulated environment, as discussed next.

2.3 Firmware Re-Hosting

A large body of work has looked at ways to run firmware in a controlled software environment on a host machine-in effect, creating a virtual replica of the embedded system [11, 74, 75]. This is often termed firmware re-hosting, and it enables the use of standard fuzzing techniques (instrumentation, fast restart, etc.) without needing the physical device for each test case. Re-hosting is appealing because it can dramatically speed up fuzzing (by orders of magnitude, since an emulator can be reset or checkpointed quickly in memory) and allows deep introspection (e.g., full code coverage measurements, memory watchpoints, or complex program analyses) that would be hard on the device. The tradeoff, however, is fidelity: how closely does the emulated firmware execution match real hardware? Wright et al. emphasize that fidelity (both in execution timing and in I/O data behaviour) is the critical concern in re-hosting, yet is very hard to quantify or guarantee [16]. In practice, re-hosting frameworks make various approximations. They may model only the CPU and some core peripherals, while ignoring or partially simulating others. This can lead to firmware running but not necessarily doing anything meaningful if it waits on an unmodeled hardware response. A classic result in this space by Muench et al highlights that in fuzzing an embedded device, a memory corruption might not cause an immediate crash in the emulator (due to absent hardware feedback), yet would still be a serious bug on the real device [76]. Thus, a key goal in emulation-based fuzzing is to model enough of the hardware to exercise the interesting code paths, without having to reimplement the entire device.

Full-system emulation: The most direct approach is to use a full-system emulator such as QEMU to simulate the microcontroller CPU and as many peripherals as possible. QEMU and similar simulators (e.g., Renode [77], or Simics [78] for high-end targets) provide a variety of device models and can run unmodified firmware images for certain platforms. For example, QEMU has definitions for ARM Cortex-M cores and some common microcontroller boards; a firmware built for one of those boards can be loaded into QEMU and executed as if on that microcontroller. Coverage instrumentation can be added by instrumenting the QEMU translated code (AFL++ and others have QEMU modes for user-space, extended by [43], and there are patches to get coverage from system-mode QEMU as well). The benefit of full-system emulation is that, if all required peripherals are modelled, the firmware sees a complete environment and can potentially run as is. In practice, though, firmware often interacts with custom or undocumented peripherals (sensors, radios, timers, etc.) that QEMU does not support out-of-the-box [79]. Early frameworks like Firmadyne targeted Linux-based IoT firmware by replacing the kernel and handling syscalls, which works for high-level code but not for bare-metal logic. For bare-metal firmware, researchers turned to a mix of static analysis and stub implementation to handle peripherals.[80] One approach is peripheral modelling: creating functional models for the hardware registers that the firmware interacts with. The models need not capture full hardware detail; they just must respond in a way that keeps firmware running. A notable example is P^{2}IM, which automatically classifies memory-mapped I/O registers into categories (control, status, data, etc.) by observing firmware's access patterns, and then supplies generic responses during emulation (e.g., fuzzer-provided random data for data registers, or dummy status flags). P2IM's strategy is to let the fuzzer itself effectively "model" the peripherals by providing input bytes whenever the firmware reads from a device register, thereby exploring different hardware behaviours without an explicit model [6]. This allowed high-throughput fuzzing of many firmware samples, discovering bugs in USB stacks, sensor handling, etc., although certain complex devices (DMA controllers, intricate timing-dependent peripherals) were beyond its scope.

Another peripheral modelling approach is HALucinator[81], which targets firmware that was written against a Hardware Abstraction Layer (HAL) API. Many embedded vendors provide HAL libraries (for example, a function `HAL_UART_Transmit()`to send bytes over a UART). HALucinator replaces these HAL calls with ”emulated” versions that simulate the hardware's effect or simply mark the operation as successful. By intercepting calls at the function level, HALucinator achieves function-level fidelity-it does not execute the actual low-level driver code, but it ensures the higher-level logic sees expected behaviours (like ”transmit succeeded”). This works for code that uses the HAL, but not for firmware that directly pokes hardware registers (the latter requires a lower-level modelling like P2IM). A more recent tool, Fuzzware by Scharnowski et al. [82], significantly improved automatic peripheral modelling by using static analysis on firmware to identify device register accesses and their usage patterns, and then generating models with appropriate semantics (e.g., modelling an ADC peripheral by a simple linear conversion). In evaluations, Fuzzware achieved higher coverage and found more bugs than P2IM for several firmware programs, highlighting the benefit of more precise modelling [82]. Still, no modelling approach is universal: certain firmware behaviours (like waiting for a hardware interrupt or relying on precise analogue sensor data) are hard to model and can cause the emulated firmware to get stuck or diverge from reality.

Hardware-in-the-loop (peripheral proxying): An alternative to modelling a peripheral in software is to forward peripheral accesses to the real hardware device. This creates a hybrid system: the CPU of the firmware is emulated, but whenever the firmware tries to read or write a memory-mapped device register, the operation is sent over a link to an actual device or component that performs it and returns a result. In other words, the emulator ”calls out” to real hardware for help. This approach is commonly called peripheral proxying or hardware-in-the-loop (HIL) emulation. A classic example is Avatar [19], which orchestrated execution between an emulator (running the core logic of the firmware) and a physical device (handling specific I/O that the emulator can't support). Avatar set breakpoints on I/O instructions in QEMU and diverted those to a proxy program that communicated with the real device via JTAG. Subsequent frameworks like Avatarr^2 extended this concept into a flexible platform for dynamic analysis, allowing analysts to choose which parts of execution run on the host vs. hardware. In fuzzing use-cases, HIL proxying can enable the fuzz target to run many instructions quickly in the emulator, but still get accurate responses for critical hardware interactions (e.g., reading an actual sensor value from the real sensor) [83]. Researchers have demonstrated fuzzing of complex embedded software by this method: for instance, Ferret combined Avatarr^2, the PANDA emulator, and the Boofuzz fuzzer to fuzz a USB firmware, forwarding USB controller register accesses to a real controller and successfully triggering memory corruptions in the firmware [84, 85, 74]. Hybrid approaches like this can achieve very high overall fidelity-since the real hardware is in the loop for peripherals, the emulator no longer needs a perfect model [86]. Wright et al. note that using real hardware for peripherals effectively gives perfect data fidelity for those components. The cost is increased complexity and reduced speed: every interaction incurs communication overhead (e.g., USB or network latency between emulator and device) [60], and one must have and maintain the physical hardware for the peripheral. SURROGATES (Koscher et al.,) even inserted an FPGA in the loop to accelerate the proxying of hardware requests, illustrating the engineering effort sometimes needed. Nonetheless, peripheral proxying is a powerful technique when full virtualization fails-it "bolts in" real hardware only for the pieces that resist emulation [87].

Recent research prototypes have refined hybrid re-hosting in various ways. Charm forwarded mobile device driver I/O to real hardware over USB, but required instrumenting the driver code (limiting it to open-source drivers) [88]. ICSemu/ICS-fuzz by Tychalas et al. [89] focused on industrial control systems: it ran PLC control logic in an emulator and intercepted calls to I/O instructions, simulating basic sensors/actuators or allowing interactive inputs, thereby enabling fuzzing of PLC programs for logic bugs. Another line of work, Jetset [90], combines static analysis and symbolic execution to target specific code regions in firmware that are hard to reach. Jetset can determine what peripheral inputs are needed to drive execution to a chosen ”goal” (like a particular function) and then essentially guides the fuzzer by providing those inputs, rather than relying purely on random exploration. This helps mitigate path explosion in the presence of many peripheral states. In a similar vein, Zhou et al., presents Emu that mixes concrete and symbolic execution to handle peripheral inputs: it runs the firmware concretely in an emulator but on each unknown hardware read, it invokes a concolic execution to solve for an input that will satisfy the firmware’s subsequent path constraints [91]. In effect, Emu can compute what device data would exercise a new path, rather than guessing. These advanced techniques improve the ”smartness” of peripheral modelling by borrowing ideas from concolic (hybrid) fuzzing-they reduce the reliance on blind random values for hardware inputs.

Sandboxing and API-level re-hosting: Some fuzzing efforts choose a middle ground in fidelity by extracting specific components of firmware and running them in isolation on the host [92]. For example, one might identify a parsing routine in the firmware (for a network packet or file format) and compile it into a Linux binary, then fuzz it with libFuzzer or AFL. This requires some adaptation-the function might need dummy replacements for hardware interactions-but can leverage the full power of user-mode fuzzers. FirmCorn by Gui et al. [93] is a framework that automates this sandboxing: it uses static analysis to cut out a firmware function along with its dependencies and builds a Linux-hosted test harness for it. By sandboxing at function-level, FirmCorn avoids dealing with the whole OS or device state; it focuses on one algorithmic piece of the firmware (say a crypto function or a packet parser). The upside is very fast fuzz iteration and easy use of sanitizer tools (ASAN, etc.) for bug detection. The downside is that many embedded bugs emerge only when the function is in its real context (e.g., misuse of a hardware buffer, or sequencing issues between threads). Thus, sandboxing is best for certain classes of logic bugs that are self-contained. Eisele et al. [40] categorize such approaches under ”sandbox emulation" of firmware. They note that while sandboxing can be effective (and several serious vulnerabilities have been found this way), it inherently ignores some interactions with hardware and other firmware parts, so its coverage of the whole system behaviour is partial.

Table 2.1: Comparison of Firmware Rehosting Techniques

Technique	**Approach**	**Advantages**	**Drawbacks**
Rehosting	Full system emulation (e.g., QEMU)	• Full control over execution • Debug-friendly environment	• **1300%+ performance overhead** • Requires hardware-specific peripheral models • Tight coupling to target architecture • Poor non-standard peripheral support
Para-rehosting	Partial emulation (shadow layer)	• Reduced emulation scope • Faster than full rehosting	• Cannot fuzz middleware components • Incomplete system state tracking • Misses hardware-middleware interactions • Shadow layer accuracy dependency
HiL	QEMU + physical hardware I/O	• Real hardware peripheral fidelity • Accurate I/O timing	• **Not scalable** (requires device pool) • High synchronization latency • Risk of physical hardware damage • Limited parallel execution
On-Device	Native hardware execution	• Maximum execution fidelity • Zero emulation artifacts	• **Prohibitive cost** at scale • Slow iteration cycles • Limited debugging capabilities • Device management complexity

To summarize the landscape: Emulation-based fuzzing has drastically expanded what can be tested without physical devices. Tools now exist across a spectrum from low-fidelity but highly automated (e.g., feed everything random data in QEMU) to high-fidelity but more manual (e.g., connect actual hardware for peripherals), shown in table 2.1. A recurring theme in the literature is the speed vs. fidelity trade-off. Fully software approaches run millions of test cases quickly but risk missing bugs due to inaccurate models; hardware-in-loop approaches ensure real behaviour but run orders of magnitude slower and require per-target effort. Hybrid techniques and smarter modelling attempt to get better performance, but there remains no perfect solution [94]. Wright et al.'s survey of re-hosting challenges identifies 28 distinct challenges ranging from obtaining firmware binaries, through handling self-modifying code in emulators, to timing synchronization between emulated components [16]. The current state of research has made impressive progress on many of these, but some (like precisely emulating timing or complex analogue peripherals) are still open problems. Nonetheless, the field has matured to the point where integrated frameworks (such as Avatar2, PANDA, or the Intel-developed MCUemu [95]) can be used by practitioners to perform security analysis on firmware via re-hosting. As for now, it is increasingly common for new fuzzers to combine techniques: for example, a fuzzer might run the CPU in QEMU (for speed and coverage instrumentation), use a partial peripheral model (for common devices), and fall back to a live device or solver-based input generation for the trickiest interactions. This multi-modal strategy is likely to continue, as no single method suffices for the variety of embedded systems in the wild.

2.4 Abstraction-Based and Hybrid Analyses

Beyond pure fuzzing, researchers have also applied symbolic execution and other program analysis techniques to embedded firmware, either to augment fuzzing (hybrid fuzzing) or to systematically explore states that fuzzing alone might miss. These we term abstraction-based approaches, since they involve analysing the firmware at a higher level of abstraction than concrete step-by-step execution on hardware or emulator.

Pure symbolic execution: Tools like FIE [96] were early efforts to symbolically execute firmware code (FIE targeted MSP430 MCU firmware) and detect vulnerabilities like buffer overflows by exploring all feasible paths. In symbolic execution, input data is treated as symbolic variables rather than concrete values; the execution generates constraints (path conditions) and uses an SMT solver to find values that drive alternate branches. FIE could find certain bugs without any concrete runs, but it suffered from the usual path explosion and limited scalability-firmware with loops or complex peripheral interactions can lead to an exponential number of paths or require models for hardware. Inception (Corteggiani et al., 2018) [17] improved on this by integrating a KLEE-based symbolic executor with a simplified CPU emulator, and crucially allowing concrete hardware-assist: when a peripheral register was accessed, Inception could fetch a concrete value from a real device to avoid over-constraining or guessing the hardware state [97]. This technique of ”hardware access forwarding” during symbolic execution reduced the need to model the entire hardware symbolically-essentially it runs symbolic execution for the code logic but queries the physical device for any unknown input, thereby focusing the symbolic effort on relevant branches [98]. Even so, purely symbolic approaches are limited to relatively small programs or short execution traces before state-space explodes. They work best as bug detectors for specific routines.

Hybrid concolic fuzzing: A more practical approach is to combine fuzzing and symbolic execution in a complementary way. Concolic (concrete + symbolic) execution runs the program with concrete inputs (e.g. provided by a fuzzer) and simultaneously gathers path constraints; periodically, it solves some constraints to generate new inputs that steer into different paths [99]. This has been popular in general software (e.g., QSYM [100], Driller [101], etc.), and has been applied to firmware as well. Ai et al. propose a concolic testing approach for embedded binaries that supports multiple architectures by hooking into the device's debug interface: they perform the concrete execution on the real device and offload the constraint solving to the host [102]. In their setup, the device is run with a given input until a certain branch is hit; then the device is paused and the execution trace (or relevant state) is fed to a symbolic engine on the PC which generates a new input that would flip one of the recently encountered branches. The new input is then tested on the device. This way, the real device provides accurate execution, and the solver helps guide exploration. The approach found some deep bugs in wireless sensor network firmware by exploring tricky checksum conditions that pure fuzzing struggled with [103]. Another recent example is ES-FUZZ [104], which runs concurrently with a firmware fuzzer and automatically selects sequences of MMIO reads where coverage has stagnated. It then performs symbolic execution on the selected portions to generate stateful, context-aware MMIO models. These models are immediately adopted by the fuzzer to guide exploration of critical peripheral interactions, ultimately reducing the input search space and significantly improving coverage.

Overall, abstraction-based techniques like symbolic execution are powerful for systematic coverage of certain code segments (especially where the input space is structured or has checksums, magic bytes, etc.), but they require significant computational resources and/or manual modelling of environment, which limits their standalone use in large firmware. In the context of fuzzing, they are increasingly used as boosters-e.g., when a fuzzer plateaus, a symbolic analysis might generate a breakthrough input. However, even hybrid fuzzers ultimately face similar challenges as pure fuzzers in the embedded realm, because they rely on concrete execution traces and thus on a working execution environment. If the firmware cannot run far without proper hardware inputs, concolic methods also stall. Thus, these methods usually assume they have at least a partially functioning re-hosted or hardware-assisted setup to work with.

Finally, we note an orthogonal dimension of abstraction: some works abstract the hardware itself. For instance, Trippel et al. describe fuzzing actual hardware designs (Verilog circuits) like software [105], though that is beyond the scope of firmware fuzzing and more related to hardware security testing. In the firmware context, ”hardware abstraction” typically refers to using layers like HAL or OS APIs as interception points, which we discussed (HALucinator, etc.).

In summary, abstraction-based approaches enrich embedded fuzzing by enabling deeper insight into program logic (through solvers) and by compensating, to some extent, for blind spots in brute-force fuzzing. The combination of fuzzing and symbolic execution-hybrid fuzzing-has shown promise in multiple 2023 works for firmware, and we expect future fuzzers to increasingly integrate lightweight concolic components that operate seamlessly with the main fuzz loop. Still, these techniques are no silver bullet; they must be carefully applied to avoid state explosion, possibly focusing on specific subsystems of the firmware where they add the most value.

2.5 Summary

This chapter surveyed the state of embedded firmware fuzzing across four themes: core fuzzer components, on-device (hardware) fuzzing, firmware re-hosting, and abstraction-based / hybrid methods.

For core fuzzer components. We reviewed how seed selection, mutation, scheduling, feedback, and crash handling must be adapted to embedded contexts. Seeds can be packets, sensor traces, files, or action scripts; good harnesses are key to feed inputs in the right way. Most systems keep AFL-style mutations and scheduling but must handle long, stateful sequences and reliable resets. Coverage remains the main fitness signal (edge or basic-block coverage), with add-ons such as memory-error signals and path-length heuristics. Crash detection uses device faults, liveness checks, emulator traps, and sanitizers. Snapshot/restore and parallelism in emulators raise throughput; on hardware, loops and watchdogs help avoid re-flashing. The field still lacks shared benchmarks and agreed evaluation rules, which makes cross-paper comparison hard.

On-device fuzzing. Using real hardware gives perfect device behaviour but makes scale and automation harder. Recent systems exploit debug ports (e.g., SWD/JTAG) and hardware breakpoints to recover coarse coverage without binary changes; trace hardware (e.g., ARM ETM) can yield near-native coverage with low overhead; and side-channel signals (power, EM, timing) can stand in when debug access is closed. Black-box interface fuzzing (e.g., network/USB/UART) remains useful but lacks feedback. Figure 2.1 shows the central trade-off: higher fidelity on the right comes at the cost of lower speed and automation.

Firmware re-hosting. Re-hosting runs firmware on a host system to gain speed, coverage instrumentation, and deep introspection. Full-system emulation (e.g., QEMU/Renode) works when device models exist; otherwise, peripheral modelling fills gaps. We covered generic MMIO modelling (e.g., P2IM), HAL-level stubs (HALucinator), and static-analysis-driven models (Fuzzware). When software models fall short, hybrid ”peripheral proxying” routes MMIO to real hardware (e.g., Avatar/Avatar2), trading speed for accuracy. Function-level sandboxing (e.g., FirmCorn) enables very fast fuzzing of isolated routines but misses whole-system effects. Table 2.1compares these choices by approach, pros, and cons.

Abstraction-based and hybrid methods. Symbolic and concolic execution can overcome plateaus by solving path constraints in checksums, handshakes, and state machines. Pure symbolic execution struggles with path explosion and device modelling; practical systems mix concrete execution with selective solving (e.g., on device via debug links, or only for chosen MMIO reads), and feed the results back to the fuzzer. These hybrids help reach deep code but still depend on a workable execution setup.

Embedded fuzzing tries to balance between fidelity, speed, and coverage. On-device methods find bugs tied to real timing and peripherals but do not scale well. Re-hosting unlocks speed and visibility but risks model errors. Hybrids bridge the gap by mixing emulation, real hardware, and solver-based guidance. The field needs shared benchmarks, clearer metrics (bugs found over time, not only coverage), and better support for precise timing and complex peripherals.

Chapter 3

Sizzler: Sequential Fuzzing in Ladder Diagrams for Vulnerability Detection and Discovery in Programmable Logic Controllers

3.1 Introduction and Motivation

Research Question 1: *Can domain-specific learning improve mutation so that more inputs pass checks and expose deeper code in PLC workloads?*

Industrial Control Systems (ICS) underpin critical infrastructure across energy, water, transport, and defence. Historically isolated, ICS are now connected to standard IT networks and use commodity software and hardware. This shift increases efficiency but also broadens the attack surface. At the core of many ICS deployments are Programmable Logic Controllers (PLCs), which drive real-time control with strict latency bounds. Recent reports show that PLCs are frequent targets in zero-day campaigns and continue to attract research attention [106].

Most commercial PLCs compile application logic—often expressed as ladder diagrams—into proprietary firmware using vendor-defined instruction sets. Prior work has shown that typical vendor toolchains lack basic safety checks at compile time [107], leaving systems exposed to issues such as timer races, unreachable code, and hidden jumps [108]. Studying these flaws is hard in practice. File formats and firmware are proprietary, and even models from the same vendor can require different memory maps for on-chip peripherals. As a result, a generic hardware abstraction layer that supports multi-vendor emulation remains an open problem and slows progress on systematic vulnerability analysis [89].

Fuzzing is a dynamic testing method that drives a target with unexpected inputs to trigger faults and explore new paths. It has a strong track record at finding runtime bugs without prior knowledge of specific flaws. However, direct fuzzing of PLC firmware is difficult: code is closed-source, vendor-specific, and tightly coupled to hardware. Emulation is a practical alternative, but it must be faithful enough to capture the behaviour of ladder logic and the I/O events that drive it.

As outlined in Challenge C3, generic mutation strategies such as AFL's *havoc* produce many invalid inputs for specialized targets like ladder logic. These inputs fail domain checks or violate stateful constraints, causing shallow exploration and wasted compute. This motivates RQ1: learning mutations that respect domain rules so more tests pass checks and reach deeper control logic.

This chapter answers RQ1 by introducing *Sizzler*[1] (SequentIal fuZZing in LaddER diagrams), a fuzzer that learns effective, domain-specific mutation sequences for PLC workloads. Rather than apply random edits, Sizzler records sequences of mutation operations that lead to new paths, trains a SeqGAN [109] to model those sequences, and then uses the model to propose likely-successful mutations at the relevant input bytes. The aim is simple: generate more valid test cases that cross checks, maintain state, and expose deeper code.

To enable realistic execution without vendor lock-in, we build a vendor-independent emulation testbed. Ladder diagrams are authored and compiled with LDmicro[2], then run as binaries on commodity MCUs within an OpenPLC-based environment[3]. We extend QEMU to improve GPIO and I2C handling and implement Modbus over TCP to broaden peripheral coverage. This setup avoids firmware re-hosting while preserving the execution semantics of ladder logic and the I/O patterns seen in the field.

We evaluate Sizzler on a suite of ladder programs compiled to firmware and executed in our emulation testbed. To test generality beyond PLCs, we also run Sizzler on the LAVA-M and Magma benchmarks and compare against established fuzzers. Results show that Sizzler increases the rate of valid inputs and the depth of path exploration, leading to higher coverage and more bugs found. In one case, a previously unknown flaw found by Sizzler was assigned a CVE, highlighting practical impact.

This chapter contributes:

- A domain-specific mutation strategy that learns sequences of edits with SeqGAN to generate valid, high-yield test cases for ladder logic.

1. Sizzler framework: `https://github.com/7linux-0/Sizzler`
2. LDmicro: ladder diagram editing, simulation, and compilation to native firmware. `https://cq.cx/ladder.pl`
3. OpenPLC: open-source PLC runtime and emulator. `https://openplcproject.com`

- A practical, vendor-independent emulation workflow for PLC applications, including QEMU refinements for GPIO/I2C and Modbus/TCP support to better cover peripherals.
- A public dataset of synthetic ladder-diagram vulnerabilities to support repeatable research and fair comparison[1].
- Evidence that the approach generalises: on LAVA-M and Magma, Sizzler achieves higher coverage than strong baselines.
- A real vulnerability discovered by Sizzler that received a CVE identifier, demonstrating real-world relevance.

We first describe Sizzler's architecture and learning pipeline, then detail the emulation testbed and datasets. We present evaluation results on PLC workloads and general benchmarks, analyse why the learned mutations help, discuss limitations, and outline future work.

3.2 Technical Background

In this section, we systematically categorize existing state-of-the-art fuzzing approaches to emphasize the novelty of Sizzler. Table 3.1 summarizes these previous fuzzing studies, separating the core functionality into three categories: *PLC vulnerability detection*, *emulation*, and *fuzzing*.

Table 3.1: Taxonomy of related work. Key: ●= Coverage, ◗= Limited Coverage, ❍= No Coverage. The methodology employed in the organization of the columns in the analysis pertains to the various techniques related to Sizzler.

	Vulnerability detection				Emulation			Fuzzing	
Approach	**PLC target**	**Binary instrumentation**	**Symbolic execution**	**Fuzzing**	**HIL**	**Data replaying**	**Abstraction replacement**	**Havoc**	**GAN based**
μSBS [110]	❍	●	◗	❍	❍	❍	❍	❍	❍
ICSFuzz [89]	●	❍	❍	●	❍	❍	❍	❍	❍
VETPLC [108]	●	●	●	❍	❍	❍	❍	❍	❍
SymPLC [111]	●	❍	●	❍	❍	❍	❍	❍	❍
P2IM [6]	◗	❍	❍	●	❍	●	❍	❍	❍
DICE [15]	◗	❍	❍	●	❍	●	❍	❍	❍
Fuzzware [82]	●	❍	❍	●	●	❍	❍	❍	❍
HALucinator [81]	◗	❍	❍	◗	❍	●	●	❍	❍
Avatar^2 [7]	◗	●	●	●	●	❍	❍	❍	❍
GANFUZZ [112]	●	❍	❍	●	❍	❍	❍	❍	●
RapidFuzz [113]	❍	❍	❍	●	❍	❍	❍	❍	●
Fasterfuzzing [114]	❍	❍	❍	●	❍	❍	❍	❍	●
MOPT [22]	❍	❍	❍	●	❍	❍	❍	●	❍
$Havoc_{MAB}$ [23]	❍	❍	❍	●	❍	❍	❍	●	◗
Sizzler	●	❍	◗	●	●	❍	❍	●	❍

3.2.1 PLC Vulnerability Detection

Several studies have focused on the analysis of PLC security, with many using fuzzing as a method for vulnerability testing. For instance, μSBS [110] presents a static binary analysis technique to detect illegal accesses to firmware memory addresses. Moreover, ICSFuzz [89] rapidly identifies vulnerabilities in PLC programs generated using the Codesys development environment. VETPLC [108] verifies real-world PLC programs to detect code safety violations through the use of static and dynamic analysis. SymPLC [111] employs symbolic execution to test both single-task and multi-task PLC programs. These studies emphasize the importance of code-level analysis in uncovering vulnerabilities and demonstrate the effectiveness of fuzzing in achieving this goal. In comparison, Sizzler supports a wider range of architectures through the use of LDmicro and OpenPLC [115] to provide internal feedback from the PLC's runtime. Additionally, we have designed Sizzler specifically for PLC Ladder Logic Diagram (LD) since is the most widely used PLC programming language [116], while other studies mainly focus on structured text language.

3.2.2 Emulation

Emulation and re-hosting techniques are pivotal in identifying and mitigating vulnerabilities within embedded system firmware. Noteworthy frameworks like P2IM [6] and μEmu [91] leverage this approach to record peripheral inputs without affecting firmware operations. Fuzzware [82] integrates an instruction set emulator with a fuzzer, supplying inputs for hardware accesses at the Memory-Mapped I/O (MMIO) registers. HALucinator [81] employs abstraction replacement using Avatar^2 [7] and QEMU [117] to substitute hardware abstraction layer calls with customized implementations. Hardware-in-the-Loop (HIL) further adds a layer of fidelity validation by simulating real-time interactions between controllers and peripherals. Specifically, HIL creates real-time virtual environment that mimics the actual physical system. When the controller sends signals to peripherals, HIL generates and returns simulated values, replicating what would occur if the controller were interfacing with real devices. Sizzler uniquely utilizes HIL in conjunction with Avatar^2 [7] to emulate Modbus, an industry-standard network protocol.

3.2.3 Fuzzing

Fuzzing stands as a widely employed automated testing methodology utilized for the purpose of vulnerability discovery. Generative Adversarial Networks (GANs) represent generative models primarily designed to produce new samples that replicate a learned distribution from a training dataset [118]. Various adaptations of GANs have been implemented in the context of automated testcase generation for fuzzing. For instance, GANFUZZ [112] utilizes GANs to learn protocol grammars for testcase generation, while RapidFuzz [113] leverages WGAN-GP to optimize the seed distribution, thereby enhancing code coverage. Nichols et al. also propose the use of GANs for augmenting the seed pool, introducing a notable mutation strategy [114]. It is worth noting, however, that GANs often face challenges when dealing with sequence data.

The current GAN-based fuzzer concentrates on the generation process. In contrast, Sizzler focuses on the mutation strategy within fuzzing, building upon the foundation of American Fuzzy Lop (AFL) [21]. Specifically, Sizzler employs havoc scheduling, a highly stochastic process, which introduces alterations to the target code through a suite of operators. These operators encompass bit flips, byte flips, arithmetic operations, and value replacements. Test cases are generated by stacking multiple operators, with the number of stacks determined randomly (AFL originally sets the maximum operator stack size to 128). This objective is motivated by the varying efficiencies observed in mutation operations, as exemplified by MOPT [22], which employs particle swarm optimization to enhance fuzzing efficiency. Additionally, $\boldsymbol{Havoc_{M}AB}$ [23] adopts algorithmic approaches for operation selection. Sizzler distinguishes itself by uniquely employing SeqGAN to learn

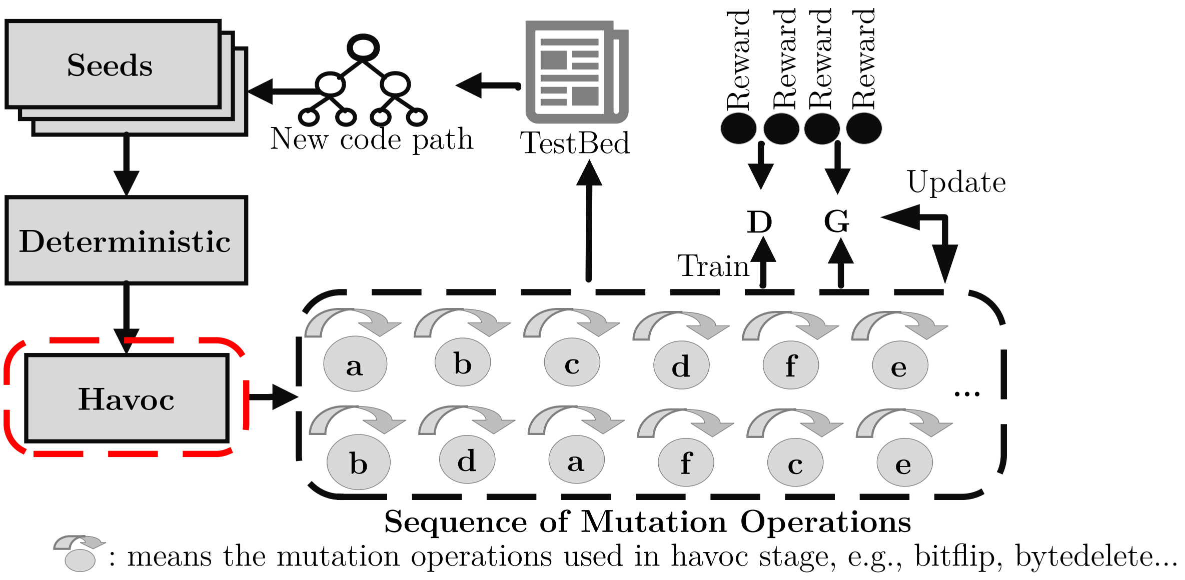

Seeds
New code path
TestBed
Reward
Reward
Reward
Reward
Deterministic
D
G
Update
Train
Havoc
a
b
c
d
f
e
...
b
d
a
f
c
e
Sequence of Mutation Operations
: means the mutation operations used in havoc stage, e.g., bitflip, bytedelete...

Figure 3.1: Sizzler architecture indicating the enhanced mutation-based fuzzing strategy using updated sequences resulted by SeqGAN training.

the optimal sequence of operators, thus improving the generation of efficient test cases. SeqGAN employs a Long Short-Term Memory (LSTM) neural network as its generative model (G) and incorporates a discriminative model (D) to differentiate between authentic and generated data.

3.3 Sizzler overview

3.3.1 Sizzler Overview

As illustrated in figure 3.1, Sizzler is engineered to generate a diverse set of test-cases aimed at identifying vulnerabilities within PLC application code, executed over emulated MCU firmware. This process is further detailed in figure 3.2. Sizzler records the input/output of the PLC program, capturing both the ability to execute the target code for each test input and any associated order-related operators. When the input produces deeper code paths or uncover new ones, the relevant combination of operations is recorded as a new dataset.

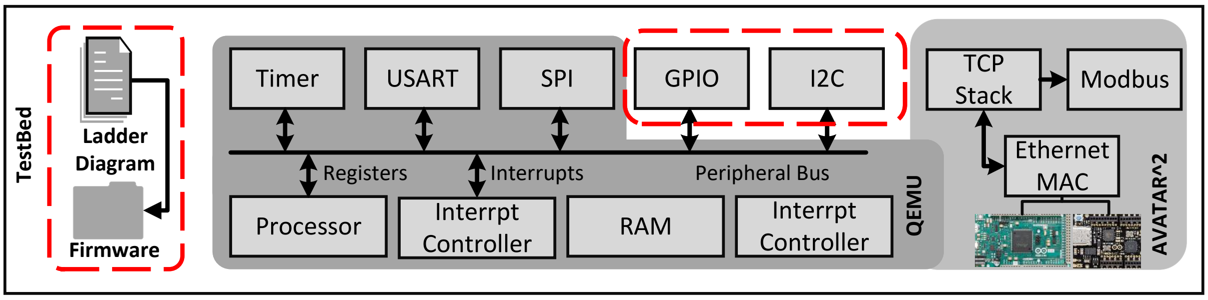
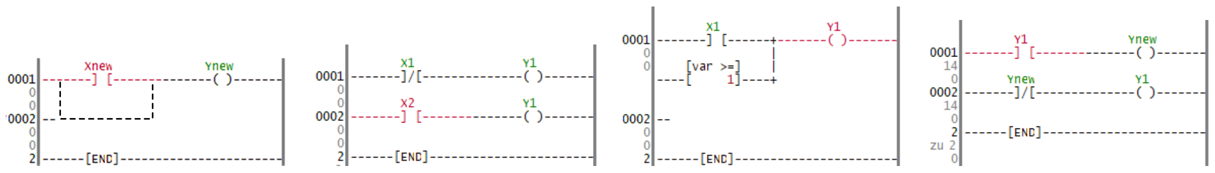


Figure 3.2: Emulation approach for assessing Sizzler fuzzing over converted ladder diagrams.

Sizzler employs the effector map to document the position of the related bits in testcases. Subsequently, SeqGAN formulation is trained using the logged inputs and the order of the recorded operations. Our implementation consists of a generator that generates sequences of operations, and a discriminator to distinguish between real and generated data, whereby the SeqGAN model is continuously updated through incremental learning. Subsequent to the training process, sequences of operations are generated to dictate the generation of new testcase inputs of converted PLC application binaries over the emulated MCU firmware.

Given the strict limitations of emulating proprietary PLC firmware, and in order to adequately evaluate Sizzler, we convert LD code into ANSI C code and customize Avatarr^2 and QEMU to emulate PLC functionality. As illustrated in figure 3.2, our implementation enables refined GPIO and I2C interfaces, such as I/O modules and communication interfaces, with commonly unsupported peripherals by QEMU. Furthermore, we utilize Avatarr^2 to emulate an HTTP server and provide Modbus/TCP communication. Thus, we provide support for PLC control applications to run on various MCU architectures (such as ARM Cortex-M and AVR ATmega) used during the evaluation phase.



(a) Hidden jumper (b) Object repeat reference (c) Comparator hardcode (d) Race condition competition

Figure 3.3: Typical ladder diagram vulnerabilities.

3.3.2 Vulnerability Composition

Ladder logic is a visual programming language that was originally designed to resemble electrical relay diagrams. A ladder program is drawn as two vertical rungs with one horizontal rung. Each rung is a small rule: it reads a set of inputs (contacts) and updates outputs (coils). Contacts behave like switches that can be open or closed, and coils represent actuators or internal bits [119].

At run time, the PLC repeatedly scans the program from top to bottom. In each scan cycle it evaluates the rungs from left to right, using current sensor readings and internal memory, and then writes the results back to outputs. This scan-based execution model makes ladder logic behave like a synchronous, cyclic program: values are sampled, processed, and applied in discrete steps. Timers, counters, and latches extend this model with simple stateful elements, but the underlying representation remains a graph of rungs rather than conventional control-flow constructs such as loops and functions. These differences matter for Sizzler, because they affect both what counts as a valid test input and how faults in PLC applications manifest during execution.

Current PLC LD compilers do not have the capability to detect vulnerabilities or intricate logic errors caused by logic injection as seen in real incidents [120]. We leverage such missing capabilities to construct 30 PLC binary applications. Each PLC application is deliberately generated with various types of vulnerabilities present in the LD program. We compose the following vulnerabilities within the LD programs:

- **Race condition competition**: occurs when two processes concurrently request the same resource. In the context of LD programs, two logical operands are executed simultaneously and race against each other, resulting in an unexpected output even though the input is the same. As shown in figure 3.3d, the output value of y_{new} is changed from 0 to 1 within two cycles, even though the inputs are fixed.
- **Missing certain coils or outputs**: A rung missing a specific output coil, such as Output Energise (OTE), latches or sets, unlatches, etc., can lead to a dependency issue where other tag(s) are impacted.
- **Infinite loop**: The main PLC application execution process is a continuous cycle. If the LD contains infinite loop, it can consume excessive CPU resources to cause PLC to crash.
- **Hard-coded logical comparator**: embedded into the application, which can consequently be accessed by attackers. Such hard-coded instructions and values can be obtained from the PLC through reverse engineering and then modified to manipulate the operation of the PLC program. As illustrated in figure 3.3c, *var* can be changed thus the whole application will be affected.
- **Missing jumps and links**: Jumps and links may not be executed following the control flow. An attacker could identify these memory addresses and utilize the spaces to insert malicious code.
- **Hidden jumpers**: can utilize the jump mechanism in PLC to skip some elements, shown as figure 3.3a. If the jumper is coded to bypass a single element within a given rung, it is possible that more than one element or even a whole branch will be abandoned.
- **Object repeat reference**: This can occur when one output may be controlled by different inputs. In LD, some operands, for OTE, timers, and counters, could have different results that depend on the scanning of different rungs with similar logic. From example, the *y1* output coil in figure 3.3b is duplicated within the LD, and will be de-energised depending on which rung is executed, resulting in an undesired output.

- **Unused objects**: It is possible that some variables remain unused, especially in large-scale PLC programs, which will not be detected by the compiler. Open and pre-instantiated entry points present a potential vulnerability in the system. This vulnerability arises from the ease with which an attacker can insert malicious code into the system.

The vulnerabilities we implement may cause severe damage to a real ICS setup. For instance, attackers can gain root authority and implant backdoors to monitor and control PLC behaviour. Moreover, gaining access to hard-coded PLC applications can allow attackers to obtain sensitive information, such as temperature and pressure values within a variety of critical industries, resulting in unpredictable consequences [121].

3.3.3 Ladder Diagram Conversion to ANSI C

We construct our emulation testbed by embedding the vulnerabilities discussed earlier into LD projects. The challenge relates to the conversion of projects into executable binaries in order to emulate their application-level characteristics. We therefore utilize the open-source LDmicro compiler and OpenPLC to transform the projects into C programs that could be compiled and executed as binaries. The compilers for OpenPLC and LDmicro are capable of defining values and addresses used by PLC pins. The OpenPLC can also map the Modbus address space directly to the physical I/O. The process of generating C code comprises three stages:

1. Lexical and syntax checks of a LD;
2. Compiler generates symbol tables such as globally declared functions, Program Organization Units (POUs), and identifiers declared for enumerated types;
3. Analysis of the executed control flow and data type to annotate the abstract syntax tree and generate C code.

The generated C file outlines the PLC runtime, initiated by establishing an array of communication-related functions in accordance with the memory map. This array encompasses both peripheral and inter-process functions, which are instantiated as threads. The LD is subsequently loaded during runtime where the instructions are then executed. The I/O modules defined in the memory map play a crucial role in receiving both analogue and digital signals, and serve as a medium for fuzzing to generate inputs for the PLC program.

3.3.4 MCU Emulation

As already mentioned, the QEMU open-source and cross-ISA emulation platform was used to address security testing challenges pertaining to control binaries. QEMU can emulate several CPUs, for example x86, PowerPC, and ARM, through the dynamic binary translation technique. However, QEMU does not support all the peripherals for different types of targets, such as I/O modules, which means our PLC binary files can not be native emulated by QEMU. We substitute low-level I/O interactions with high-level implementations to enable external interaction and emulation of PLC firmware over five different MCUs. We achieve this by customising and mapping crucial board-level communication protocols of QEMU, such as GPIO and I2C, over specific MCU memory address regions.

We emulate the GPIO controller to capture varying sensor signals represented within the underlying physical process controlled by the PLC; for instance, switch closures and button presses. GPIO is a type of digital signal pin that is integrated into the circuit and can be set as an input or output. By default, GPIO ports often have no pre-defined task, however the pins can be customized and controlled by software to achieve desired functions. In our work, the PLC's I/O interfaces receive sensor signals and relay them to the GPIO interface. The GPIO then stores the received data in its memory-mapped space. By means of a specialized function, we retrieve data from the GPIO device file

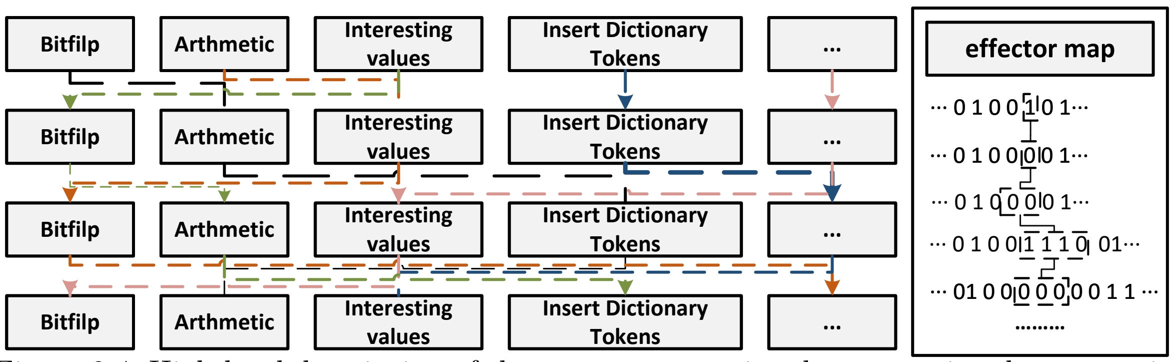


Figure 3.4: High-level description of the processes associated to capturing data mutations within the Sizzler havoc process.

through a read system call and transfer it to the memory space of the control process. This operation is carried out by a thread that is created alongside the control process and employs a write system call to transmit the input data. The sequence of events is repeated at a frequency determined by the QEMU scan cycle length of the control application.

A PLC uses I2C functions to interface with peripheral devices, such as sensors and actuators. We use I2C to connect to the GPIO, emulating PLC board-level communication. The I2C communication protocol requires the SCL (Serial Clock Line) and the SDA (Serial Data Line) wires to communicate between the runtime and the GPIO. The microcontroller acts as the I2C master, managing the signals as well as sending and receiving data over the SDA line.

The emulation of GPIO and I2C is programmed on the interface layer and is based on the same logic where a driver is created for each interface. Since we use firmware based on three different programming boards, different datasheets are required to record all registers for the GPIO and I2C. The different drivers are then mapped into memory areas, which initialise the device and I/O ports and select the register to be written or read according to the offset from the device's address. For example, to emulation logic of GPIO for ARM STM32F40X, where its memory region is *0x3FF*. We define every register's status and trace from *0x40020000* to *0x40022c00*. The trace change of GPIO is stored in log files to identify which register is being accessed, assisting with the monitoring of the emulation process. To successfully configure alternative MCUs, the GPIO drive address in the memory map and the range of addresses of registers need to be adjusted according to the datasheets. The initialisation process for I2C communication on MCUs is similar to GPIO. Notably, the main difference from GPIO is that I2C opens log files and performs a read system call to emulate an SDA line, thus moving input data to its own memory space.

The Modbus protocol is extensively utilized to enable Master/Slave industrial communication between PLCs and other ICS components. In our refined implementation, the remote master initiates read and write requests to the OpenPLC slave sending Modbus frames over the network (Modbus/TCP). Emulation of Modbus via the TCP stack is performed using the QEMU emulator that we have refined within this work, while communication is achieved through Avatar^2 and is tailored for different MCU boards. However, the binary application generated by OpenPLC can only act as a slave. During operation, the runtime leverages the TCP stack to translate messages to Ethernet frames, which are subsequently dispatched via the physical Ethernet port using an Ethernet library, such as tuxeip.

3.3.5 Sizzler Enhanced Fuzzing

Sizzler builds upon the original AFL fuzzing approach, and by contrast, implements a mutation approach that is sensitive on code branches to discover deeper code paths related to a vulnerability. The mutation strategy employed in Sizzler, presented in algorithm 1, uses a customized havoc approach as seen in AFL. We enhance our approach by using SeqGAN to increase the number of useful test cases to be used during the fuzzing process and optimize the havoc stack process. Specifically, the sequence of strategies enable an increased amount of edges and solves the context-insensitive problem. In the first fuzzing cycle, we collect the dataset by recording sequences of operations that find new code paths.

```
Algorithm 2: Sizzler fuzzing
Input: seed; fuzz_one(); fuzz_time_counter $t \leftarrow 0$
1  Select a seed input;
2  Initialise a queue of inputs to be processed;
3  Execute mutation strategies on queue;
4  if save_if_interesting(seed) then
5      if Is_Havoc() then
6          Push the series of strategies into data_reader() when fuzzing is executing
            havoc();
7          Choose the matebyte in effector map;
8      end
9  end
10 while $t \bmod 10 = 0$ do
11     Initialise the generator and discriminator;
12     Train the Seq_Gan() model;
13     Update the generator based on reward;
14     Generate new strategies;
15     $t \leftarrow t + 1$;
16 end
17 common_fuzz_stuff(): Execute new strategies on interesting seed and matebytes;
```

The dataset is then forwarded to the SeqGAN model to generate new strategies. figure 3.4 illustrates how different operations would match with each other. For example, "*Bitflip* → *Havoc* → *Insert Dictionary Token* → *Interesting values*" represents how to formulate the execution path through different operations.

On the right part of figure 3.4, a list of binary patterns illustrates how input values are permuted by different operators. The list of operations that could trigger a new execution path is stored in the query. Meanwhile, the position of related bytes is recorded in the `effector map`. The effector map serves as a guide for the havoc process whereas bytes causing different code paths are called matebytes. In general, we observe that in the context of the inherently data-intensive AFL, bytes originating from the same section of the input data frequently induce identical code paths during execution. Thus, in Sizzler, a byte is designated as a matebyte only if its alteration results in an execution path that is distinct from the paths generated by modifying adjacent bytes. Because we observed that bytes originating from the same section tend to lead to the same code paths. During the execution of the havoc process, the matebytes encoded in the effector map are subject to modifications by the operators. In the subsequent cycle, Sizzler employs SeqGAN to simultaneously train both a generative model and a discriminative model. These models utilize the values stored within the effector map to generate new testcases.

The generative model continuously refines its strategies based on the rewards it receives, enabling it to iterate effectively on mutation attempts. Consequently, even if initial mutation attempts are unsuccessful on a new seed, the model remains capable of discovering effective mutations in subsequent iterations. It's noteworthy that SeqGAN is not limited to a single mutation strategy; rather, it possesses the capability to synthesize a diverse portfolio of strategies. This diversification significantly enhances the likelihood that at least some of these strategies will prove effective when applied to new seeds. A salient attribute of SeqGAN lies in its ability to dynamically adapt and optimize its mutation strategies over time, facilitated by its integrated reward mechanisms. The process of training SeqGAN contains the following steps:

1.Training Data Capture and Pre-processing: The sequence of operators that induce variations in the code path are captured as effective data points for training, recorded as $x = (x_1, x_2, x_3, ...x_n)$. The maximum stack size for mutators in the AFL framework is set at 128, which is the rationale for selecting 128 as our batch size. To standardize the training process, certain efficient sequences of operations are appended with specialized characters. For filling gaps in the sequence, the most frequently occurring operators within the current data are used. Subsequently, Min-Max scaling is employed to normalize these sequences, converting them into standard decimal data with temporal features.

2. Model Construction, Training and Validation: In our architecture, the generator and discriminator are defined with four layers. The generator receives a 100-dimensional noise vector $(y_1, y_2, y_3, ...y_n)$, sampled from a Gaussian distribution, as its input. The architecture includes two LSTM layers, each consisting of 128 units, followed by a Dropout layer implemented with a rate of 0.3 to mitigate the risk of overfitting. The generator's learning rate is meticulously calibrated at 0.001 for optimization purposes. Subsequently, the discriminator is trained on data generated by the Generator. Comprising three layers, the discriminator's primary objective is to differentiate between real and generated sequences, as demonstrated in equation 3.1.

$$\begin{aligned}\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)]+\\ \mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))]\end{aligned} \tag{3.1}$$

The output of the discriminator serves as a reward signal for each generated sequence, and policy gradient methods are deployed to update the generator based on these estimated rewards, as defined in equation 3.2. Here, $J(\theta_G)$ is the expected reward for the generator, θ_G are the parameters of the generator network, $p(x \mid \theta_G)$ is the probability of a sequence x given the generator's parameters, and $R(x)$ is the reward for sequence x. The generative model is then updated based on the reward to improve the quality of the generated data. To monitor the model's performance, we employ a validation set. Early stopping is triggered if no improvement in the loss metric is observed over a span of 100 epochs.

$$\begin{aligned}\nabla_{\theta_G} J(\theta_G) &= \mathbb{E}_{x\sim p(x|\theta_G)}[\nabla_{\theta_G} \log p(x \mid \theta_G) R(x)]\\ &= \mathbb{E}_{x\sim p(x|\theta_G)}[\nabla_{\theta_G} \log p(x \mid \theta_G) \sum_{t=1}^{T} \gamma^{t-1} r_t]\end{aligned} \tag{3.2}$$

As shown in algorithm 1, the AFL is firstly executed to record the different combination of strategies that trigger new code paths in the havoc process. We then utilize the SeqGAN model to capture the logic from different sequences of strategies. Similar series of operations are then generated to mutate the seed set in the subsequent fuzzing cycles. The new test cases are generated to feed into the GPIO's port to test the security of PLC application binaries. The strategies are recorded throughout 10 cycles. Subsequently, the dataset is then erased and recollected through the mutation steps in order to retrain the model.

Grammar based fuzzers are highly effective when the target input language is stable and well specified, as a formal grammar can enforce syntactic validity by construction. However, for PLC ladder logic, this approach is impractical. Ladder programs are typically authored in vendor specific languages and compiled into proprietary binary formats. Even within the OpenPLC framework, the translation from ladder diagrams to C code, and subsequently to firmware, relies on toolchains that evolve over time. Consequently, a general grammar for ladder logic firmware would need to model not only the visual language structure but also compiler translations, vendor specific extensions, and the stateful behaviour of timers, counters, and inter-rung interactions. Developing and maintaining such a grammar would entail significant manual effort and inevitably tie the system to specific vendors-undermining our goal of a vendor, neutral PLC fuzzing framework.

Instead, Sizzler employs a strategy that learns to mutate existing ladder based testcases, ensuring they remain valid while exploring deeper control logic. SeqGAN is particularly well-suited to this objective: it treats mutation steps as a discrete series of actions and utilizes sequence modelling to capture long-range dependencies between edits. In contrast to simpler baselines, such as fixed mutation schedules or n-gram models, SeqGAN can adapt its mutation policy based on observed reward signals (such as coverage and bug discovery) without requiring prior knowledge of the underlying grammar. SeqGAN can also learn the dependence between different mutator operators instead of generating a single mutation strategy directly. Adopting a sequence of mutation operators can help generate more valid testcases based on interesting seeds even when the precise structure of the ladder firmware is unknown or only partially observable.

3.4 Evaluation

3.4.1 Research Question

Research Question 1: *Can domain-specific learning improve mutation so that more inputs pass checks and expose deeper code in PLC workloads?*

This evaluation focuses on two primary aspects: (i) assessing whether Sizzler can generate valid testcases and achieve high code coverage on PLC ladder diagram applications; and (ii) determining whether it demonstrates improved coverage and bug detection capabilities compared to baseline fuzzers on general benchmarks.

3.4.2 Evaluation Methodology

Testbed and Datasets: Our evaluation was conducted over a server equipped with an AMD Ryzen Threadripper 3960X 24-Core Processor with 64GB of RAM and Geforce RTX 3050 graphics card, running Ubuntu 18.04. We constructed 30 vulnerable PLC control binaries generated through the conversion of LDs. Each binary was programmed to perform different control system functions such as time measurement and traffic light control, which are then implemented on five MCUs[4]. The LDs were originally acquired from GitHub and several projects were tasked with utilizing them in real-world production environments[5]. The diagrams underwent secondary development, which involved integrating various vulnerabilities into the binary code, as defined in Section 3.3.2.

To demonstrate how generalisable Sizzler is to non-ICS specific environments, we also evaluate using the Large Volume Automated Testing (LAVA-M) dataset, which comprises of four GNU coreutils programs (uniq, base64, md5sum, and who) [122]. Moreover, the LAVA-M dataset has been widely used as a benchmark for other fuzzing evaluations [26, 123] to evaluate their performance, enabling us to compare with state-of-the-art approaches. The LAVA technique was employed to create a ground-truth baseline by including a substantial number of realistic bugs within the binary source code. Each bug was assigned a unique identification number displayed upon activation. Additionally, we conducted a comparative analysis of Sizzler's performance against other prevalent fuzzing tools, utilizing Magma version 1.2 as the testbed [27]. Magma serves as an expansive repository of targets modeled on real-world computing environments. It comprises seven distinct libraries and 16 executable binaries. Contrary to LAVA-M, which relies solely on artificially synthesized bugs and magic byte comparisons, Magma offers a diverse range of vulnerabilities that are categorically aligned with the Common Weakness Enumeration (CWE) framework. In total, Magma encompasses 138 identifiable bugs, consisting of 15 integer errors, six of which manifest as divide-by-zero errors, as well as 58 memory overflow issues. The remaining bugs span various types, including use-after-free, double-free, and null-pointer dereference vulnerabilities. We conduct the fuzzing benchmark on two dataset for 24 hours and repeat ten times.

4. In particular the PIC1616F628, PIC1616F88, Atmel AVR ATmega 2560, Atmel AVR ATmega 128, and St ARM STM32F40X MCUs.
5. `https://github.com/BongPeav/LdMicro`

Table 3.2: The result of Unit Test

Peripheral	F103 Arduino			F103 RIOT			SAM3 Arduino			SAN3 RIOT		
	p2im	μEmu	Sizzler	p2im	μEmu	Sizzler	p2im	μEmu	Sizzler	p2im	μEmu	Sizzler
ADC	✓	✓	✓	N/A	N/A	N/A	✗	✓	✓	✓	✓	✓
DAC	N/A	N/A	N/A	N/A	N/A	N/A	✗	✓	✓	✓	✓	✓
GPIO	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓
PWM	✗	✓	✗	N/A	N/A	N/A	✗	✓	✗	✓	✓	✗
I2C	✗	✗	✓	N/A	N/A	N/A	✗	✗	✓	N/A	N/A	N/A
UART	✗	✓	✓	N/A	N/A	N/A	✓	✓	✓	✓	✓	✓

The symbol ✓signifies that the emulator has successfully passed the unit test, while the symbol ✗indicates that the emulator has failed the unit test. The notation "N/A" is employed to denote scenarios where the combination of the MCU and associated libraries is not adequately supported by real devices.

3.4.3 Unit Test for Emulation

We conducted the same unit-test experiment as was done in P2IM to ensure a head-to-head comparison, using an identical set of 44 firmware samples. These samples encompass eight MCU peripherals, and two distinct MCU chips: the STM32 F103RB, and the Atmel SAM3X8E. Each unit-test sample embodies a unique yet feasible combination of peripheral configurations. The firmware executes rudimentary operations associated with these peripherals.

A comparative analysis was conducted among Sizzler, P2IM, and μEmu. P2IM identifies processor-peripheral interfaces and provides applicable input data via these interfaces on behalf of the peripherals. Conversely, μEmu leverages symbolic execution to discern appropriate values for peripheral access and dynamically responds to read operations initiated by peripherals.

As illustrated in Table 3.2, Sizzler attained a passing rate of 82.3%, markedly surpassing P2IM's 41.1%. μEmu recorded the highest efficacy with a passing rate of 88.2%. A significant impediment to the success of unit tests within P2IM is the misclassification of peripheral registers, a consequence of categorizing these registers according to their access patterns. Such misclassifications lead to an inability for P2IM to meet the firmware's expectations, leading to stalled execution. Sizzler's suboptimal performance, when compared to μEmu, can be attributed to its inability to synchronize effectively with the emulator, resulting in firmware halts. Interestingly, both μEmu and P2IM fail to emulate binary files embedded with LD, and are deficient in detecting infinite loops. Specifically, μEmu's detection is hampered when registers within the loop possess concrete values, and is effective only when the processor context incorporates one or more symbolic values. Additionally, P2IM is ill-equipped to manage non-generic peripherals such as GPIO.

3.4.4 PLC code Vulnerability Discovery

To evaluate how effective Sizzler performs at identifying vulnerabilities in PLC LDs, we utilize the 30 vulnerable binary control applications compiled from LDs. The refined SeqGAN model collects data from the havoc stage in the first cycle and subsequently uses this data to train and generate sequences of operators to guide fuzzing in the following cycle. Each binary is subjected to fuzzing for ten cycles in order to record the entire process. The SeqGAN model is then retrained for the next cycle. Table 3.3 presents the results of these experiments, demonstrating the code coverage and the time consumption for ten cycles.

Evidently, Sizzler demonstrates execution times that typically exceed 40 minutes. We can attribute this to the simplicity of the LD logic, as well as the inclusion of hardware abstraction functions for MCUs within the PLC binaries. The control programs which use the Modbus protocol to read/write registers were exclusively developed using real PLC models (Ethernet libraries) to compile a set of applications containing vulnerabilities in order to evaluate the fuzzing capabilities of Sizzler.

Table 3.3: The code coverage result of ladder diagram for different MCUs executing converted PLC applications.

MCU	**Program**	**Vulnerability**	**Crash**	**Time(*min*)**	**Coverage(%)**		
					Function	*Basic-block*	*Edge*
PIC1616F628	test-i2c	RC	Y	64	93.8	78.1	69.5
	test-water	UT	Y	49	85.7	77.3	66.7
	test-electric	IL	Y	61	76.3	66.4	51.1
	test-i2c-lcd	MJL	Y	57	88.7	77.4	71.6
	test-leds	IL+UO+MJL	N	41	72.5	60.0	44.1
	test-pwm	UT+CH+ORR	Y	60	94.1	81.4	77.1
PIC1616F88	test-spi	ORR	Y	55	91.6	77.4	64.8
	test-uart	UO	Y	49	93.7	71.1	58.5
	test-timer	RC+UT	Y	43	91.6	67.7	51.7
	test-var-timer	UO+CH	Y	31	91.9	61.1	33.7
	test-coil	RC+IL	Y	52	87.5	78.8	67.4
	test-masterrelay	RC+UO	Y	67	94.5	81.4	71.9
AVR ATmega 2560	test-switch	MJL	Y	50	96.3	88.9	84.4
	seg-display	CH	Y	53	97.1	88.4	76.5
	asm-demo	IL+UT	Y	41	89.8	73.4	51.1
	test-blink	UO+ORR	Y	50	89.6	64.7	40.1
	test-lift	RC+HJ	Y	67	97.6	88.7	80.7
	test-alarm	UO+IL	Y	66	93.7	81.6	74.4
AVR ATmega 128	test-traffic	HJ	Y	63	91.8	81.1	76.0
	test-train	RC+IL+HJ	Y	63	99.6	89.6	84.6
	test-polution	UT+IL+ORR	Y	61	99.7	81.1	75.9
	test-counter	RC+MJL+UO	Y	31	97.4	76.7	61.2
	test-pressure	UT+IL+HJ	Y	44	82.4	71.1	61.7
	test-control	CH+RC+HJ	Y	49	88.1	63.5	59.4
ARM STM32F40X	test-clamb	IL	Y	57	87.1	71.4	66.6
	test-intustion	RC+ORR	Y	51	94.6	89.7	78.5
	test-modbus1	UT+CH	Y	61	81.6	74.4	71.7
	test-modbus2	RC+CH	Y	48	83.1	76.5	64.0
	test-tem	UT+CH+ORR	Y	55	88.0	80.0	74.3
	test-mov	RC+CH+HJ	Y	43	91.8	82.9	74.6

Vulnerability: Race condition competition (RC), Unconditional transfer (UT), Infinite loop (IL), Comparator hardcoded (CH), Missing jumps and links (MJL), Hidden jumpers (HJ), Object repeat reference(ORR), Unused objects (UO)

Furthermore, Sizzler demonstrates a high function coverage rate with nearly every function in the 30 LD programs and MCU libraries being detected under each of the five MCU architectures. Achieving an average function coverage rate of 88.4% indicates that the majority of binary functionality has been exercised. Moreover, the average basic block and edge coverage rates of 71.29% and 61.4%, respectively, provide further insight into

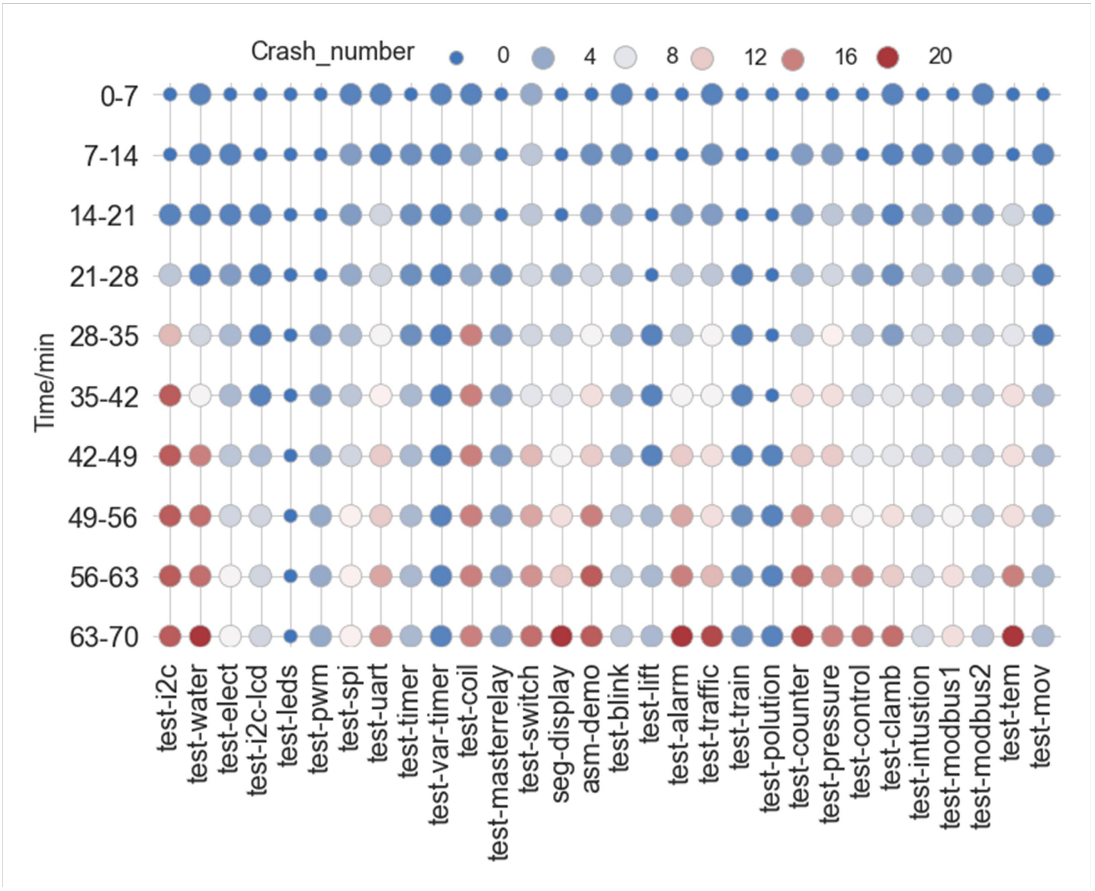


Figure 3.5: Fuzzing results for developed PLC binary applications.

the executed code paths. Sizzler detects crashes in 29 out of the 30 PLC LD programs, with the exception of `test-leds`. Sizzler also did not perform optimal basic block and edge coverage for the PIC1616F628 and PIC1616F88 architectures, which can be attributed to the less sophisticated emulation of PIC that halts if non-emulated peripherals are accessed, significantly restricting firmware execution.

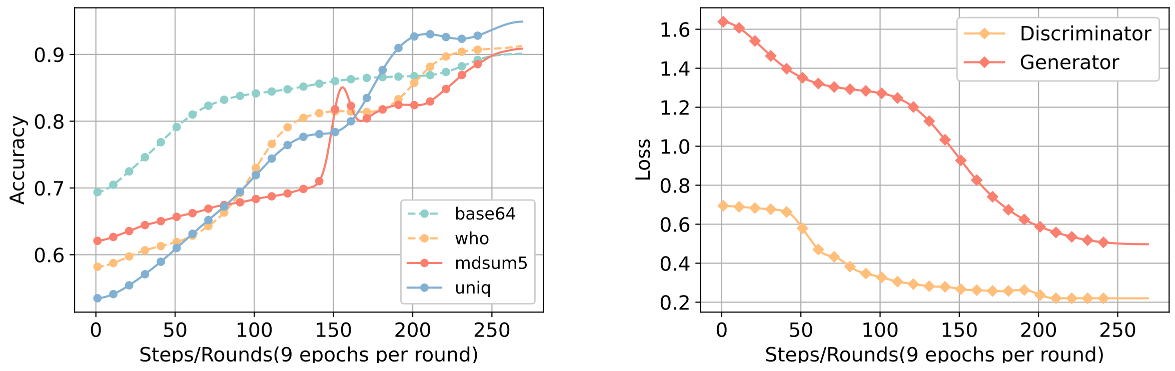
The results presented in figure 3.5 indicate that the application of Sizzler to PLC LDs yields a higher number of vulnerabilities than expected. Specifically, the analysis revealed more than 20 vulnerabilities, despite that the 30 programs were initially implemented with only one or two known vulnerabilities. One reason for this is that Sizzler uses public libraries that provide basic functionality for firmware, such as timers and UARTs. These libraries often lack appropriate access controls and fail to properly manage memory, leading to buffer overflow or other vulnerabilities. For example, the MCU library does not properly validate the input size or check for heap overflows when it converts parallel data from a microprocessor into serial data. Sizzler sets the data buffer to a high value, causing the application to crash. From the perspective of an attacker, these types of overflow vulnerabilities can be exploited by allocating a large amount of memory onto the heap and then writing beyond the end of the buffer, hence resulting in the execution of arbitrary code and unauthorized access to the device.

3.4.5 PLC Vulnerability and CVE Assessment

In order to assess our findings regarding PLC vulnerabilities discovered by Sizzler, we leverage the OpenPLC project to construct a cost-effective PLC based on both Arduino and STM32 platforms. We conduct this analysis to re-run the detected vulnerabilities and verify Sizzler's generic practicality. The discovered vulnerabilities are incorporated into proof-of-concept projects over an emulated PLC instantiated through openPLC. We select two exemplar bugs that are the most frequently detected by Sizzler.

Timer integer overflow. Sizzler successfully identified an integer overflow vulnerability that could potentially lead to an infinite loop in the executed program. The vulnerability is associated with the Ladder Diagram (LD) timer function in the code under analysis. This integer overflow is triggered when Sizzler assigns an abnormally large value to the input parameter. Consequently, the program running within the OpenPLC environment becomes ensnared in an infinite loop.

Cycle integer overflow. The Ui_Ccycle variable is used to control the behaviour of three different output coils based on whether the cycle count is within certain ranges. In each rung, if the cycle count is within the appropriate range, then the corresponding output coil is turned on; otherwise, it is turned off. Sizzler sets Ui_Ccycle to be incremented to its maximum, and it will thus wrap around 0 causing the program in openPLC to continue executing indefinitely.



(a) Accuracy Performance (b) Generator/Discriminator Loss

Figure 3.6: SeqGAN Evaluation in LAVA-M.

Vulnerability verified by CVE. Additionally, one vulnerability identified by Sizzler has been officially recognized by the Common Vulnerabilities and Exposures (CVE) system, under the identifier CVE-2023-43184[6]. This vulnerability pertains to a buffer overflow issue in the OpenPLC runtime. It enables attackers to inject malicious code via the slave_device attributes, thereby escalating to higher root privileges and inducing a server crash when the Programmable Logic Controller (PLC) establishes connections with other equipment via the Modbus protocol. An additional vulnerability was detected by Sizzler and has been substantiated in [124], CVE-2018-20818. Specifically, a buffer named in_-memory is declared in the glue_generator.cpp file. This buffer is also invoked in the modbus.cpp file and is susceptible to being overwritten beyond its 1024th position, thereby interrupting the loop and causing the runtime to halt.

3.4.6 General Vulnerability Detection

SeqGAN assessment: We evaluate the ability of Sizzler to perform vulnerability discovery using the LAVA-M dataset as seen in various studies, by firstly assessing the SeqGAN performance. SeqGAN comprises two phases: Pre-training and Adversarial Training. During the pre-training phase, the model employs the Maximum Likelihood Estima-

6. https://packetstormsecurity.com/files/174582/OpenPLC-Webserver-3-Denial-Of-Service-Buffer-Overflow.html

tion (MLE) approach to train the generator to produce a negative sample. The pre-training phase consists of 120 steps and 120 epochs. The discriminator is then pre-trained for 50 steps, with each step comprising of three epochs. Subsequently, the generator and discriminator are adversarially trained consisting of 180 rounds, where each round comprises three steps, and each step includes three epochs, in order to obtain accurate data.

Accuracy: The results presented in figure 3.6 demonstrate that Sizzler achieved high performance for establishing the path of mutation operators. As depicted in figure 3.6a, high accuracy is achieved when generating data with base64 and uniq, exceeding 90%, and model performance becomes stable after the 250th epoch. Further analysis reveals that the model achieves 80% accuracy on md5sum datasets, which is likely due to the datasets containing a high degree of redundancy, resulting in an imbalanced distribution of the data.

Loss: We observed a consistent decrease in loss after the 250th epoch, which demonstrates the stability of Sizzler. The change in loss for the SeqGAN model during the training phase can be seen in figure 3.6b. Moreover, the generator loss drops rapidly during the first ten epochs and experiences a second drop between the 140th and 150th epochs. Subsequently, both the generator and discriminator losses remain below 0.5, indicating that both are well-trained and capable of establishing sequences of operators for fuzzing. The low loss provided by the generator implies that it is effectively producing sequences that the discriminator assigns a high probability of being real. Conversely, the low loss of the discriminator suggests that it is accurately classifying sequences as real or fake.

Vulnerability discovery comparison: We compare the performance of Sizzler with state-of-art fuzzers, including AFL [21], AFL++ [43], MOPT [22], Angora [26], and NEUZZ [123]. AFL++ is an upgraded version of AFL, which uses intermediate feedback-driven fuzzing and experimental fuzzing strategies to generate testcases. MOPT use the Particle Swarm Optimization algorithm to search the mutation operators and accelerate

Table 3.4: Bugs found by different fuzzers on LAVA-M dataset.

Fuzzer	Base64	Md5sum	uniq	who
Dataset Baseline	44	57	28	2136
AFL	0	0	2	1
AFL++	3	1	19	772
MOPT	2	3	N/A	N/A
Angora	48	57	26	1531
NEUZZ	46	55	27	1562
Sizzler	44	46	18	981
Sizzler+Angora	48	57	28	1711

the speed of generating testcase. Angora uses dynamic taint analysis to track the data flow of inputs through the program and detect bugs in real-time. Angora is therefore able to quickly identify problematic inputs and focus the fuzzing process on those inputs. Conversely, NEUZZ uses a surrogate neural network for branch behaviour approximation of a target program and implements the gradient-guided technique to generate test inputs. Here we measure the amount of source code or assembly instructions that are executed, where the higher the code coverage, the greater the likelihood of identifying bugs within the code. The code coverage is collected by using `afl-cov` [125]. `Afl-cov` analyses the coverage of programs by instrumenting the binary and collecting data on which lines of code were executed to generate instrumentation codes and graphic displays to present the code coverage rate information.

The results obtained from the LAVA-M dataset were computed over a 24-hour period based on ten independent iterations of the experiment and are tabulated in Table 3.4. The results indicate that Sizzler achieves superior performance and discovers a higher number of code bugs that relate to vulnerabilities as compared to AFL, MOPT, and AFL++. However, it was noted that NEUZZ and Angora achieved higher bug identification when using the LAVA-M dataset. As the LAVA bug injection technique only injects a single bug type, for example, an out-of-bounds memory access triggered by a ”magic value” comparison, some magic bytes are not copied from the input directly but rather are computed from the input. NEUZZ employs a feed-forward neural network approach to

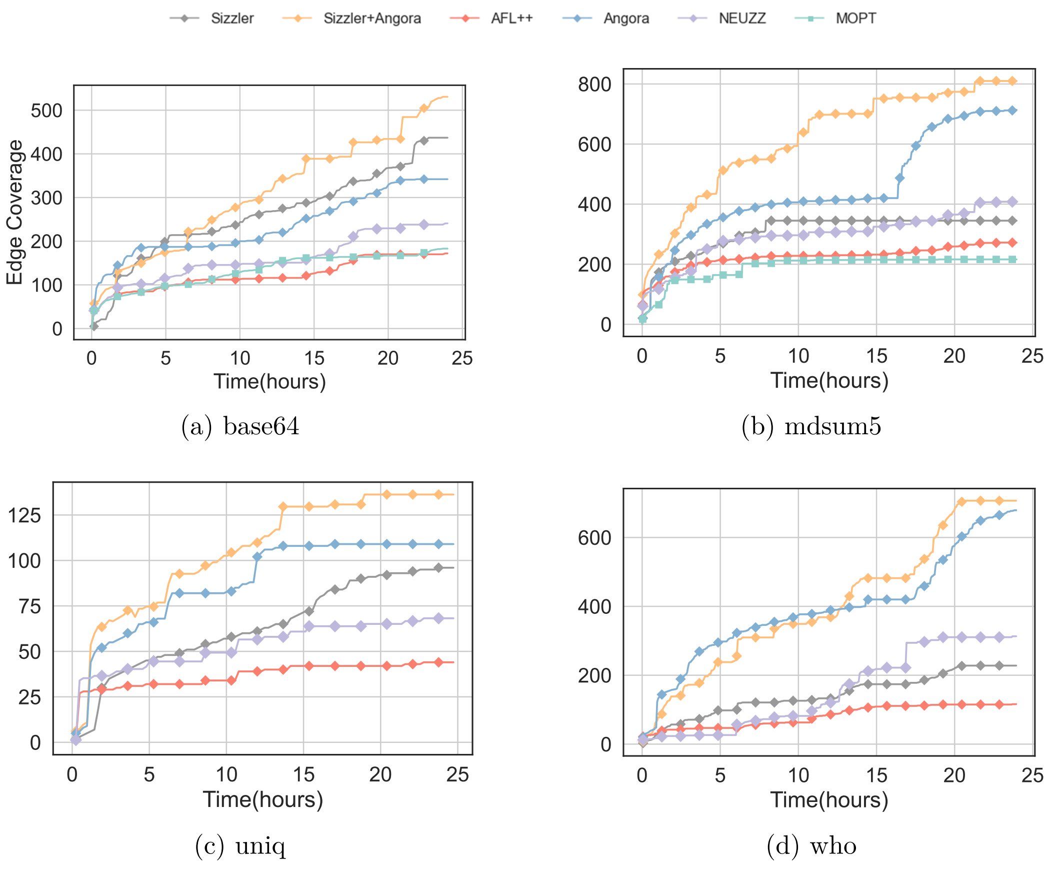


Figure 3.7: Code coverage performance across all fuzzing approaches in the LAVA-M dataset.

identify potential bytes within the proximity of a designated ”magic value”. Conversely, Angora uses a context-sensitive approach, wherein it tracks the input byte offsets that lead to a specific predicate and subsequently modifies these offsets through gradient descent, as opposed to relying solely on the concept of ”magic bytes”. Hence, we propose a novel strategy that combines our mutation technique implemented in Sizzler with Angora to facilitate the identification of magic values. The performance of this hybrid approach can be seen in Table 3.4, which illustrates that the Angora-Sizzler strategy achieves the highest performance among the five fuzzers evaluated where 129 additional bugs were identified in the target program compared to the performance achieved by the NEUZZ fuzzer.

The results presented in figure 3.7 provide a comprehensive evaluation of the edge coverage trends for the approaches under investigation over a 24-hour period of execution using various benchmarks. A thorough analysis of the data reveals that the combination of Angora and Sizzler demonstrates a superior performance, surpassing both the original

Angora and other state-of-the-art fuzzers in the LAVA-M dataset. In addition, the performance of Sizzler improves over time, likely due to the completion of the deterministic process stage and transfer to the havoc stage, which enhances the efficiency of the fuzzer. Furthermore, using SeqGAN to learn the sequence of stacked strategies and generate more efficient testcases also contributes to the improved performance of Sizzler.

Figure 3.8 demonstrates the computational performance of Sizzler by presenting the average execution speed over a 24-hour period while performing fuzzing over the LAVA-M dataset. Sizzler achieved a fuzzing throughput ranging from 0-3500 executions per second, which represents a significant increase in performance compared to other state-of-the-art fuzzers. Specifically, the range of the average throughput for other fuzzing approaches was observed to be between 0-600 executions per second. One potential reason for the significant increase in computational performance is through the use of emulation techniques, where the binary files are executed on their native architecture rather than the host architecture. In addition, the emulation process also assists in reducing the time spent on initialisation and setup tasks between test cases. Furthermore, Sizzler places a greater emphasis on the havoc stage, which results in the generation of a larger number of test cases, thereby contributing to an overall increase in the throughput of the fuzzing process.

The performance of Sizzler on the Magma dataset, shown in figure 3.9, is measured through the arithmetic mean of discovered bugs per trial per day. On average, Sizzler identified 39 bugs, while AFL++, Angora, NEUZZ, and MOPT detected 19, 17, 15, and 37 bugs, respectively. Notably, Sizzler and MOPT exhibit comparable performance metrics on the Magma dataset. However, Sizzler outperforms MOPT on libraries such as libxml2, OpenSSL, and Poppler, while MOPT shows superior performance on libpng, libtiff, PHP, and SQLite3.

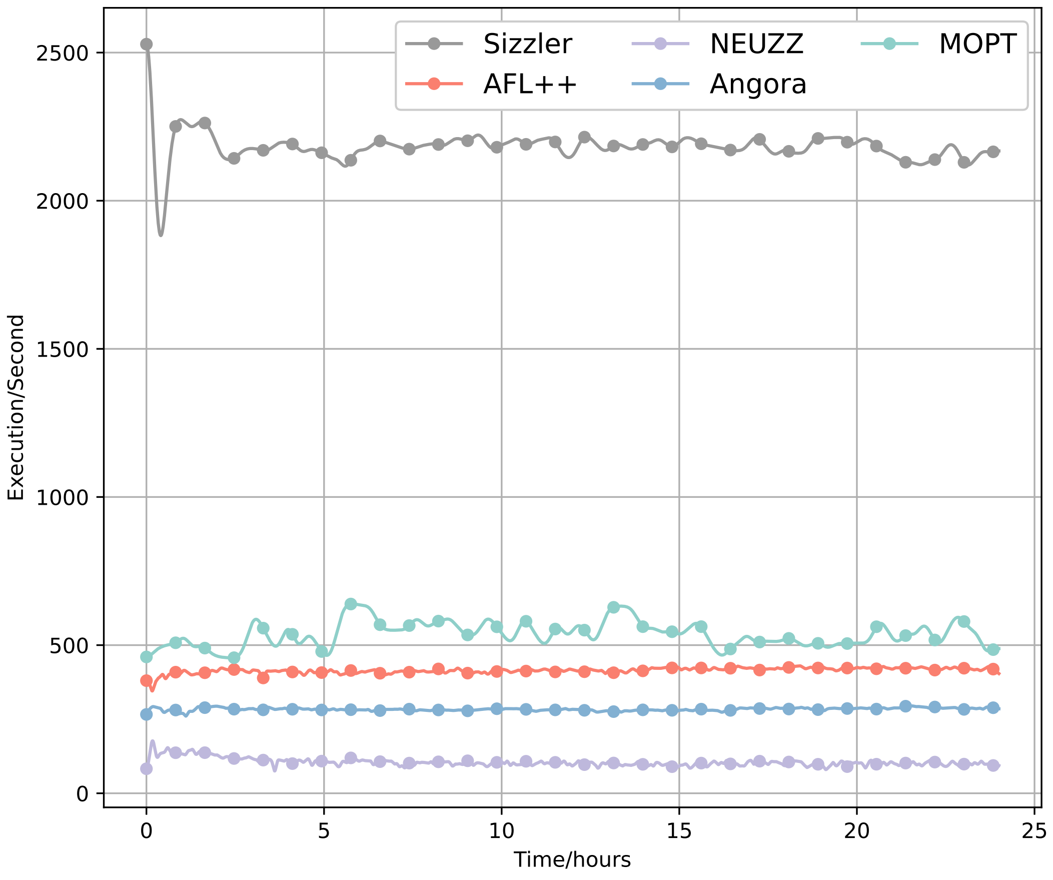


Figure 3.8: Execution speed comparison between Sizzler and other fuzzers on the LAVA-M dataset.

Though Angora demonstrates high performance on the LAVA-M dataset, it falls short of expectations on the Magma dataset. This discrepancy can be attributed to Angora's underlying assumption that target functions are continuous, thereby utilizing gradient descent for optimization. In contrast, program inputs are often characterized by byte values that are both bounded and discrete. For instance, the vulnerability PNG001 (CVE-2018-13785) in libpng is a divide-by-zero bug, triggered when the 'width' value is set to 0x55555555 and the number of channels is 3. When Angora mutates the value in proximity to 0x55555555, it is likely to calculate an erroneous gradient, thus impeding correct progress. Additionally, owing to the vulnerability of the metric to outlier influence, the capacity for drawing robust conclusions about fuzzer performance is restricted. To address this limitation, we employed a statistical significance test on the collated sample-set

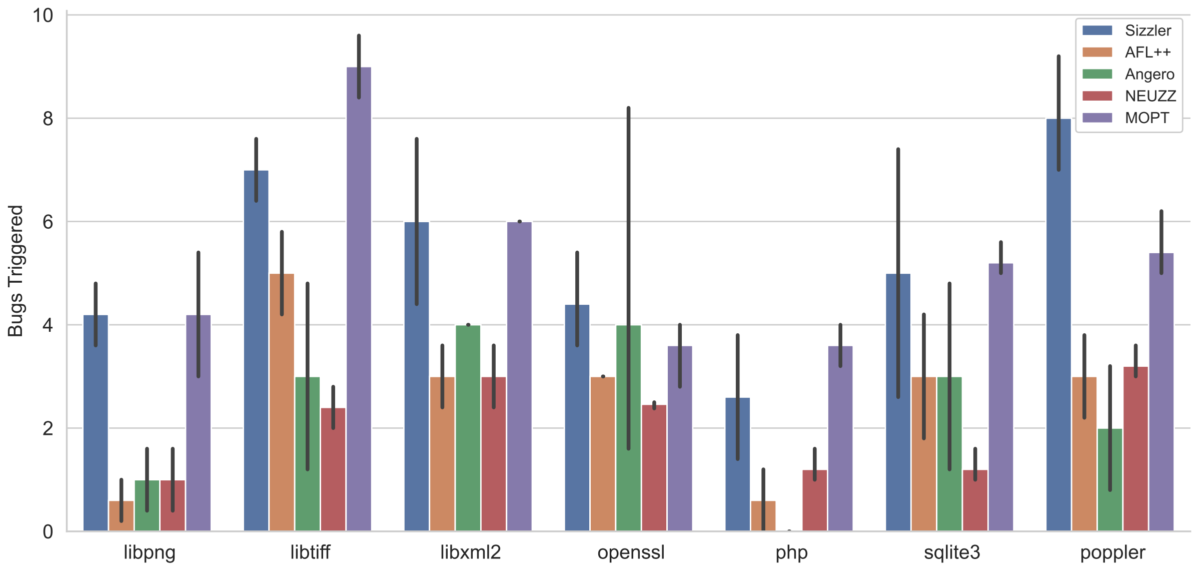


Figure 3.9: Arithmetic mean of the number of bugs found by each fuzzer across ten 24 campaigns

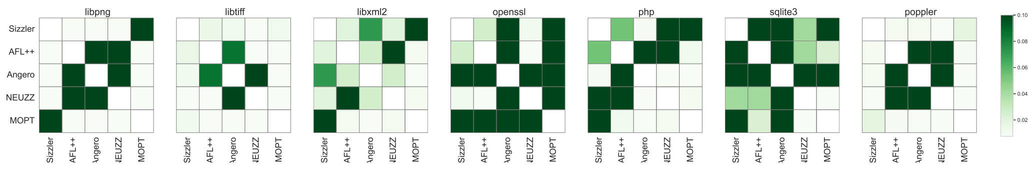


Figure 3.10: Significance of evaluation of fuzzer pairs using p-values from the Mann-Whitney U-Test.

pairs, leveraging the Mann-Whitney U-test to ascertain p-values. These p-values function as quantitative indicators for assessing the degree of dissimilarity between pairs of sample sets, as well as for evaluating the statistical significance of these disparities. Figure 3.10 presents the outcomes of this statistical significance analysis. We selected a threshold of $p < 0.05$ to evaluate the results. The analysis reveals that AFL++, Angora, and NEUZZ exhibit analogous performance against the majority of targets, notwithstanding minor variations in the arithmetic mean of discovered bugs. In contrast, both Sizzler and MOPT manifest a statistically significant enhancement in performance, outperforming all other fuzzers across seven distinct targets.

3.5 Threats to Validity

Internal Validity. One key factor affecting internal validity pertains to the reliability of our evaluation results, which may potentially be affected by random variation. To address this concern, we adhered to the methodology outlined by Klees et al. [14] on LAVA-M and Magma dataset, aiming to mitigate the influence of randomness during the assessment of fuzzers. As for the target PLC applications, we ran only a single fuzzing campaign on the corresponding MCU. Each campaign is stochastic: AFL's mutation scheduling and seed selection are randomized, and the emulation of MCU peripherals introduces further timing variation. Because we did not repeat these campaigns, the reported coverage and vulnerability findings for table 3.3 should be interpreted as a single sample rather than a stable average. We partially mitigate this by using a fixed random seed and identical configurations across targets, but we do not control for all sources of randomness in the hardware and emulation stack. A stronger design would repeat the entire fuzzing process ten times per target and report mean and variance. Since there is no baseline which supports emulation of the PLC firmware, we acknowledge that the PLC on MCU results may not be exactly reproducible, so we treat them as indicative rather than definitive.

Another concern arises during the application of the SeqGAN model for sequence generation, wherein we employ the weighted binary cross-entropy loss function to address the issue of class imbalance. Additionally, we incorporate early stopping mechanisms to monitor the validation loss and mitigate the risk of overfitting.

External Validity. In order to enhance external validity, the primary concerns revolve around the selection of subjects and benchmarks. In response to these potential threats, we have undertaken a deliberate approach. Specifically, we have chosen four well-regarded hybrid fuzzers and two recently published emulators from esteemed software engineering and system security conferences. Furthermore, in our evaluation process, we have incorporated both the Magma dataset and the LAVA-M datasets. These measures have been implemented to bolster the external validity of our study.

Validity Construction. The question of construct validity in this research primarily hinges on the utilization of edge coverage as a surrogate for code coverage. To mitigate this concern, we've adhered to a rigorous methodology. Specifically, we've employed afl-cov, an integrated tool within the AFL framework, for systematic edge coverage data collection, in line with best practices established in the fuzzing community [26, 123]. Additionally, to provide a comprehensive evaluation of fuzzing effectiveness, we have incorporated the metric of unique crash counts. These methodological choices aim to strengthen the construct validity of our study.

3.6 Limitations & Future directions

Regardless of the promising outcomes proposed through Sizzler, we argue that vulnerability discovery in PLCs, as well as embedded systems more generally, remains a huge challenge. We provide two main limitations of the proposed solution and briefly discuss potential future avenues for research:

1. Sizzler is intended for analysing PLC binary applications. However, such applications are created in diverse formats that are vendor-specific. Moreover, the programming languages for PLCs are tailored to meet specific requests of different vendors. Hence, a dynamic analysis of these binaries is contingent on a case-by-case approach,

precluding the possibility of a universal approach. Even OpenPLC project provides access to source code for PLC binary application. The vulnerabilities triggered by Sizzler for PLC can not be triggered in commercial PLC. Additionally, there have been specialized efforts to employ fuzzing techniques targeted at PLC equipment, notable among them being ICSFuzz and VETPLC. It should be noted, however, that these tools are vendor-specific and do not offer a comprehensive benchmark. Consequently, a broader performance comparison of commercial PLC for Sizzler is currently unavailable. Overcoming this limitation through further advancements would broaden the applicability of Sizzler for analysing PLC binary applications.

2. Our approach to firmware emulation is subjected to a significant challenge as vendors often restrict access to technical datasheets required to establish a suitable development environment. In addition, testing embedded firmware in real-time, whether on target devices or through emulation, is a time-consuming process. Furthermore, the AFL testing framework embedded within Sizzler is hindered by substantial tracing overhead, which leads to a significant performance impact of nearly 1300% for binary-only programs when operating in QEMU mode, as reported in [20]. Future work efforts can be directed towards improving the overhead of fuzzing techniques whilst targeting full-stack testing with high fidelity for PLCs and other embedded systems.

3.7 Conclusion

PLCs are core building blocks for numerous mission-critical ICS however they are not equipped yet with adequate mechanisms focusing on vulnerability assessment nor discovery. By contrast with wider embedded systems or MCUs, PLCs have not been extensively studied due to the intrinsic restrictions related to emulation of their firmware and proprietary application-level properties. In this chapter, we introduce *Sizzler; a PLC vulnerability discovery framework underpinned by a novel mutation-based fuzzing strategy instrumented* *over SeqGAN and PLC firmware emulation setup approach.* Sizzler is the first to achieve the translation of PLC LDs into C code, which execute on representative MCUs such as to emulate as realistically as possible a variety of PLC firmware environments across 30 PLC applications. Moreover, the optimal synergy of a SeqGAN formulation with a havoc-based mutation strategy for fuzzing through Sizzler demonstrates high efficiency on detecting new and deeper code paths that relate to an increase of discovering otherwise unseen PLC code vulnerabilities. In parallel, Sizzler is also successfully deployed and assessed within a wider embedded systems dataset associated to non-PLC applications indicating its superiority over commonly used fuzzing schemes.

Chapter 4

FuzzRDUCC: Fuzzing with Reconstructed Def-Use Chain Coverage

4.1 Introduction and Motivation

Research Question 2: *Can reconstructed def-use chain coverage provide more useful feedback for fuzzing binaries than traditional edge coverage?*

In the previous chapter we improved mutation (RQ1) so that more inputs pass checks and reach deeper code. The guidance, however, still relied on control flow coverage (edges/b-locks). For firmware and drivers this signal can be too coarse: it shows which blocks execute, but not whether important values flow to the places that matter.

RQ2 asks whether dataflow feedback can guide fuzzing better. Dataflow coverage tracks definition – use (def-use) pairs: it records when a value written at one site reaches its uses elsewhere. This aligns the feedback with program semantics rather than only control structure, and can steer inputs toward driver and peripheral code that edge coverage alone may miss.

We present FuzzRDUCC, a binary-only dataflow coverage mechanism. It reconstructs def-use chains from binaries with Angr and instruments QEMU's TCG to update a def-use bitmap at runtime. FuzzRDUCC integrates with AFL++ without changing the rest of the pipeline, so we can compare dataflow coverage against standard edge coverage under the same conditions.

We evaluate three aspects of RQ2: feasibility (can we extract and track def-use on binaries), cost (runtime overhead versus edge coverage), and effectiveness (new paths and bugs found). We report the cases where dataflow feedback helps and the trade-offs when overhead limits throughput.

To address RQ2, we introduce a methodology that emphasizes dataflow tracking in binaries without debug symbols. We use the Angr [126] framework to extract and select def-use chains according to a simple heuristic, and we integrate this feedback into execution to provide precise guidance to the fuzzer.

4.1.1 The Fuzzing for Binary

American Fuzzy Lop (AFL) [21] and AFL++ [43] have gained widespread recognition within the research community as a quality baseline for fuzzing research. Numerous studies have developed their methodologies based on AFL's capabilities. AFL is a grey-box fuzzer that generates test cases using a variety of mutation strategies tailored to achieve comprehensive code coverage. For binary fuzzing, AFL incorporates QEMU [117], a generic and open-source machine emulator and virtualizer, to emulate the execution of binaries. This emulation facilitates the addition of instrumentation, enabling AFL to obtain feedback from the binary's execution to obtain the binary's control flow. When new code coverage is discovered, AFL adapts its mutation strategy based on the test case associated with this coverage.

4.1.2 Towards Dataflow Coverage

```c
/* If the first 4 bytes are 0x01f401f4 (udp src and dst port =
   500) we most likely have UDP (isakmp) traffic */
if (tvb_get_ntohl(tvb, 0) == 0x01f401f4) {
    protocol = TCP_ENCAP_P_UDP;
} else {
    protocol = TCP_ENCAP_P_ESP;
}
if (g_ascii_strcasecmp(header_name, "Content-Length") == 0) {
    // Process Content-Length
} else if (g_ascii_strcasecmp(header_name, "Transfer-Encoding") ==
    0) {
    /* Process Transfer-Encoding header and other headers */
}
```

Listing 4.1: simple Code Example

While code coverage is a powerful tool in fuzzing, it has shortcomings when dealing with data-intensive program constructs. In Listing 4.1, an `if` statement from Wireshark checks whether the first four bytes of a packet match the specific magic number `0x01df401f4`, indicating UDP traffic (specifically ISAKMP). Code coverage can only indicate if this condition is true or false. However, when the condition fails, code coverage does not reflect how close the input is to the target value. Without proper guidance, the fuzzer must blindly guess the correct value, facing a probability of success of 1 in 2^{32}[127].

Two common strategies to address such branches are concolic execution [100][128] and intelligent branch solving [129][130][26].

Concolic execution models constraints as symbolic expressions, allowing solvers like SMT solvers to find solutions[131]. By treating input bytes as sequences of 8-bit vectors, we update their symbolic representations during execution. However, constraint solvers often struggle with simple string comparisons. Although concolic execution can systematically explore program paths to solve these constraints, it cannot differentiate between meaningful and superficial path differences. For instance, the function `g_ascii_strcase-cmp` performs case-insensitive comparisons. Different headers like `Content-Length` and `Transfer-Encoding` result in distinct paths, even when header order changes, leading to path explosion and resource exhaustion.

Intelligent branch solving struggles with this issue without manual intervention. Since comparing a single character can easily succeed, the branch is quickly marked as solved, and further analysis is skipped. As a result, constraints for the remaining characters never reach the solver. Modern fuzzers attempt to mitigate this with program specific optimizations. For example, AFL++ uses *CmpLog* instrumentation to record operands of failed comparisons and applies heuristics to solve them [132]. Instead of instrumenting branches in a general way, it relies on a hard coded list of comparison functions, treating each call as an abstract branch clearly non scalable approach.

To improve these methods, dataflow coverage provides a more precise approximation of program behaviour by focusing on how variables are assigned and used, rather than just the sequence of executed operations. This approach considers more complex structures instead of solving constraints, like lookup tables, binary trees, and directed graphs, offering deeper insights into program execution. By shifting from a control flow to a dataflow perspective, fuzzing techniques can be made more effective [133].

4.1.3 Def-use Chain Analysis

A def-use chain links a specific variable definition to all subsequent uses that it can reach without being overwritten by an intervening definition. These chains make data dependencies within a program explicit, concretely representing how a value assigned to a variable propagates to later execution points. The utility of these relationships was recognized early in compiler design. Kildall's work introduced a unified iterative framework for dataflow problems, laying the groundwork for systematically computing reaching definitions using lattice theoretic fixpoint algorithms [134]. By the mid 1980s, standard texts such as the "Dragon Book" had formalized def-use chains as a central concept in static analysis, providing algorithms to compute them for various compiler transformations [135]. This foundational research established that, while determining exact dynamic def-use relations in arbitrary programs is undecidable, a conservative static approximation of all possible pairs can be computed via iterative analysis. In practice, compilers safely over-approximate these chains to capture every potential dependency, a requirement for sound optimization.

Def-use chains have also played a significant role in software testing, particularly in the development of dataflow coverage criteria. Rapps and Weyuker introduced criteria that require test cases to cover specific def-use pairs within a program [136]. The underlying intuition is that standard control flow coverage may miss faults that manifest only through specific computations or data interactions. By ensuring that, for each variable definition, at least one test drives execution along a path to one of its uses, testers increase the likelihood of exposing erroneous computations. For example, the *all-uses* criterion mandates that for every definition of every variable, the test suite must include at least one path where that definition reaches each of its possible use sites. This ensures that the flow of values from definitions to consumers is explicitly executed and verified. Empirical studies have demonstrated that dataflow testing can be more effective at detecting certain classes of bugs than pure control flow coverage, as it forces the exercise of value propagation rather than mere branch traversal. Modern testing tools and static analysers continue to rely on computing def-use chains via static analysis similar to that of a compiler to identify critical pairs, slice programs for impact analysis, and detect anomalies such as uninitialized variables [137].

The evolution of def-use analysis is directly reflected in modern compiler infrastructures. Production compilers, such as GCC and LLVM, incorporate decades of research on dataflow analysis, making def-use relationships an integral part of their intermediate representations. LLVM, for instance, represents programs in Static Single Assignment (SSA) form [138] and provides APIs to efficiently traverse use-def chains for any given value. Each LLVM IR `Value` object maintains a list of its uses, a design that reflects the necessity of fast def-use queries, as many compiler passes frequently need to identify the consumers or definitions of a value to perform transformations. Optimizations such as common subexpression elimination, register allocation, and vectorization all benefit from these rapid lookups [139]. Furthermore, the explicit def-use links in SSA simplify alias and dependency analysis. When reasoning about memory accesses, compilers construct def-use chains not only for registers but also for memory locations, abstracted via memory SSA or alias graphs to represent memory dependencies [140].

Recently, the integration of dataflow information has enriched the traditional coverage guided fuzzing paradigm. Classic fuzzers, such as AFL, focus primarily on control flow coverage (e.g., discovering new basic blocks or edges). However, edge coverage does not directly indicate whether a fuzzing input has exercised a critical data dependency, for instance, whether a value produced at one point successfully influences a later check. Research has begun to incorporate def-use chains as a feedback metric to address this limitation. Mantovani et al. [141], for example, introduced a technique that tracks data dependency coverage, effectively measuring the number of unique data flows exercised by generated inputs. By instrumenting the program to record when a definition is used downstream, their fuzzer rewards inputs that cover new def-use chains rather than just new control flow edges. This approach guides exploration toward states that require satisfying specific data conditions. Similarly, hybrid fuzzers like Angora [26] and TaintScope [142] utilize dynamic taint analysis. While they do not explicitly construct static def-use chains, they share a similar goal: tracing how input bytes flow to affect key program points (such as branch conditions) to solve constraints.

FuzzRDUCC is designed to apply these principles to binary only firmware, drivers, and libraries. Unlike previous approaches that insert instrumentation at compile time, we reconstruct def-use chains from disassembled binaries and use angr to resolve definition and use sites. We then hook QEMU's dynamic translation to emit coverage events whenever a selected definition reaches its corresponding use at runtime. This design allows dataflow guided fuzzing to be applied even when only stripped binaries are available, exploiting the key advantage of dataflow coverage: rewarding test cases that propagate critical data toward security sensitive operations, even when those operations lie along already covered control flow paths.

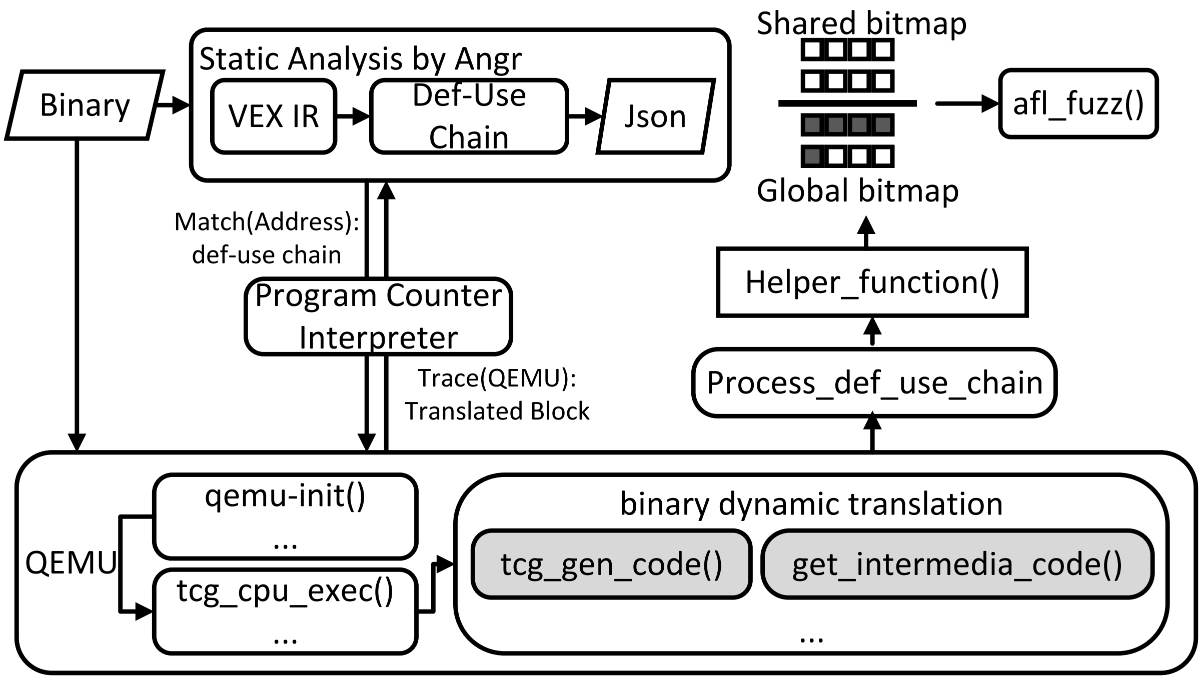


Figure 4.1: Structure of FuzzRDUCC

4.2 FuzzRDUCC Overview

Our approach enhances fuzzer effectiveness by incorporating def-use chains, structured into two main phases: static analysis and fuzzing (see Figure 4.1). In the static analysis phase, we use the Angr framework to extract def-use chains from the binary, obtaining precise addresses and counts of defs and uses for each translated block. This involves instrumenting the code to record the addresses and numbers of defs and uses, leveraging QEMU's lightweight code generation.

In the fuzzing phase, we repurpose the AFL++ bitmap (previously used as a proxy for edge coverage) to monitor the coverage of def-use chains accurately. As each basic block executes, we update the local AFL++ bitmap against a global map to track changes in execution state. This mechanism guides the fuzzer to re-mutate inputs based on analysis of previous seeds, aiming to significantly improve fuzzing efficiency by combining static analysis with dynamic fuzzing.

4.3 Methodology and Implementation

4.3.1 Def-Use Chain Generation

We extract def-use chains from binaries through symbolic execution using the Angr framework, departing from traditional methods that rely on Static Single Assignment (SSA) form. Angr loads the binary components, including library dependencies, and uses VEX Intermediate Representation (IR) to reconstruct the control flow graph and dataflow graph directly from machine code. This process maps out the program's execution flow and provides a representation of all possible execution paths, enabling comprehensive analysis of the binary's execution semantics [143].

Our method incorporates reaching definition analysis [144] to determine where variables (definitions) are assigned values and where these values are used across different basic blocks. This analysis reveals relationships between definitions and uses in the code, identifying uses reachable from definitions that have not been overwritten, using an over-approximation strategy. While this may sacrifice some soundness, it offers increased speed in analysing binaries, which we consider acceptable for achieving sufficient precision in binary-level analysis.

By storing the def-use chains in a JSON file, we facilitate their integration into QEMU's code generation process during dynamic binary translation. By systematically identifying and analysing def-use chains in binaries, we lay the groundwork for more effective fuzzing strategies by enhancing our ability to uncover vulnerabilities through understanding dataflow. This methodology outlines potential pathways through which definitions affect uses, providing a solid foundation for tracking interrelations and dependencies within the code. By enabling a focused exploration of the software's execution space, it enhances the precision and efficiency of fuzzing processes, thereby improving vulnerability detection through a thorough understanding of the software's internal mechanisms.

4.3.2 Code Instrumentation

After reconstructing the def-use chains, we integrate them into the execution of the binary managed by QEMU, which decomposes the binary into basic blocks. Each block is translated into a host-specific block through QEMU's Tiny Code Generator (TCG), converting each instruction into micro-operations within the translated block. During dynamic binary translation, these instructions are transformed into host instructions tailored to the specific architecture.

We adapt QEMU to utilize its tracing capabilities to obtain information about translated blocks, specifically retrieving the Program Counter (PC) value for each executed block. The TCG functions as a just-in-time compiler, translating guest instructions into executable code for the host architecture. By retaining the guest PC for each block and employing a hash table to associate it with the host PC of the translated block, we achieve precise tracking of control flow and dataflow.

The translation process in QEMU is divided into a frontend and a backend. The frontend lifts target instructions into TCG Intermediate Representation (IR), which is stored in a list. We focus on tracing the current execution of translated blocks (TBs), particularly utilizing the cache list to identify the current TB and obtain its PC. We then correlate the PC with the def-use chains stored in the JSON file generated by Angr, mapping the def-use chain within each translated block.

With the def-use information for the binary, we apply precise instrumentation to monitor identified definitions and uses within these blocks. After acquiring the def-use chain for each translated block, the backend converts the TCG IR into host machine code. TCG IR registers are categorized into various types: global, local temporary, normal temporary, fixed, constant, and extended basic block (ebb). Our objective is to encapsulate definitions and uses within TCG registers, generating corresponding IR to embed into the translated block for recording purposes.

We utilize helper functions to pass parameters to registers and execute jumps to specific addresses. These helper functions can also access the CPU environment, enhancing our ability to manipulate and track the execution flow.

This approach mirrors the afl_maybe_log function used in AFL++, which inserts IR into the translated block to monitor execution. However, our instrumentation focuses on usages rather than recording every definition and usage, recognizing that in statically compiled binaries, some definitions may not be utilized or analysed correctly. Focusing on usages is critical for understanding data manipulation.

By integrating def-use chain information with QEMU's execution trace, we gain deeper insights into execution patterns, facilitating more targeted fuzzing to uncover vulnerabilities. This dynamic tracking of data and control flow enables precise identification and analysis of critical execution paths and enhances our ability to detect and assess the impact of definitions and uses throughout the software's operation.

4.3.3 Optimizing Def-Use Chain Selection

Instrumenting all def-use chains in translated blocks introduces significant time and space overhead during fuzzing. For example, in the binutils dataset, one binary's translated block size increased fivefold after instrumentation [133]. To mitigate this overhead, we propose a heuristic algorithm that selectively targets addresses of common external library functions and optimizes the def-use chain selection process.

To reduce overhead, we exclude definitions and uses within the same block or function, thereby reducing the size of the translated blocks. We also disregard definitions that are not used or not detected by Angr, streamlining the analysis process. By calculating the distance between definitions and their related uses, we focus on definitions and uses that span across different functions, utilizing interprocedural analysis to efficiently identify the necessary def-use chains.

Angr's simulation involves instruction emulation and symbolic execution for branch decisions, maintaining stacks of states with register values and memory addresses. State duplication can lead to explosion, especially in loops dependent on user input, causing delays in vulnerability detection. To optimize analysis, we use Angr to identify addresses of common libc functions such as `malloc`, `calloc`, and `free`, avoiding detailed examination of external library functions. We employ Angr's SimProcedures to replace third-party library functions with custom implementations that simulate behaviour, which is important for statically compiled targets where analysing external libraries is resource-intensive.

We designed custom hook functions (`handle_malloc`, `handle_calloc`, `handle_free`) for SimProcedures to simulate memory management effects on the analysis state. For example, if `malloc` is at address `0x400900` in the binary, Angr hooks this address with its SimProcedure for `malloc`. When execution reaches `0x400900`, the SimProcedure is invoked instead of the actual `malloc`, allowing efficient reaching definition analysis on specific addresses while focusing resources on primary binary analysis.

This heuristic captures ”interesting” def-use chains, increasing the likelihood of discovering new crashes and exploring more code paths. It enhances the efficiency of our instrumentation and improves the overall effectiveness of our fuzzing strategy.

After identifying def-use chains within a target binary, we introduce an alternative coverage bitmap to track changes in these def-use chains. This bitmap records runtime relationships between definitions and uses, logging any changes observed.

When a modification in this bitmap indicates a change in dataflow coverage, we initiate a strategic re-mutation of the seed. This re-mutation aims to explore unexplored code paths, broadening coverage and deepening the fuzzing process. This method ensures a nuanced and dynamic examination of the binary’s behaviour, enhancing the potential for identifying vulnerabilities.

4.3.4 Updating the Coverage Scheme

AFL++ uses QEMU’s TCG IR to insert instrumentation code that computes a hash for each edge during execution. An ”edge” represents a transition between code blocks (e.g., from block A to block B). For each transition, AFL++ generates a unique identifier i by hashing the addresses of both source (A) and destination (B) blocks, with the source address right-shifted:

$$i \leftarrow \text{addressof}(B) \oplus (\text{addressof}(A) \gg 1). \tag{4.1}$$

This hash i indexes into the edge coverage bitmap, where each index represents a potential execution edge. When an edge is traversed, AFL++ increments the value at that index, enabling it to monitor executed edges and prioritize inputs that explore new paths.

We adapt AFL++'s tracking to capture the relationship between definitions and uses. At code generation time, we use precomputed def-use chains (from Angr's JSON files) for each translated block and employ a helper function to embed def-use instrumentation into the TCG IR.

Since every block has definitions, inserting IR into every block can slow translation. Therefore, we focus on usages, tracing them to identify related definitions. We document def-use edges according to the def-use chains. Our revised hash function for the coverage bitmap index uses the addresses of the definition and use sites:

$$i \leftarrow \text{addressof}(\mathit{def}) \oplus \text{addressof}(\mathit{use}). \tag{4.2}$$

Using these addresses as hash values generates a unique i, reducing collision risk. This approach allows tracing multiple definitions and uses within a single block, providing nuanced and sensitive coverage feedback. It enhances analysis granularity and improves fuzzing efficiency by focusing on critical dataflow aspects of program execution.

4.4 Preliminary Evaluation

Research Question 2: *Can reconstructed def-use chain coverage provide more useful feedback for fuzzing binaries than traditional edge coverage?*

To answer RQ2 we study three aspects: (i) feasibility, i.e., whether we can reconstruct def-use chains and instrument binaries at scale; (ii) cost, i.e., how the additional instrumentation affects fuzzing throughput; and (iii) effectiveness, i.e., whether def-use guided fuzzing improves coverage and crash discovery compared with edge coverage guided fuzzers.

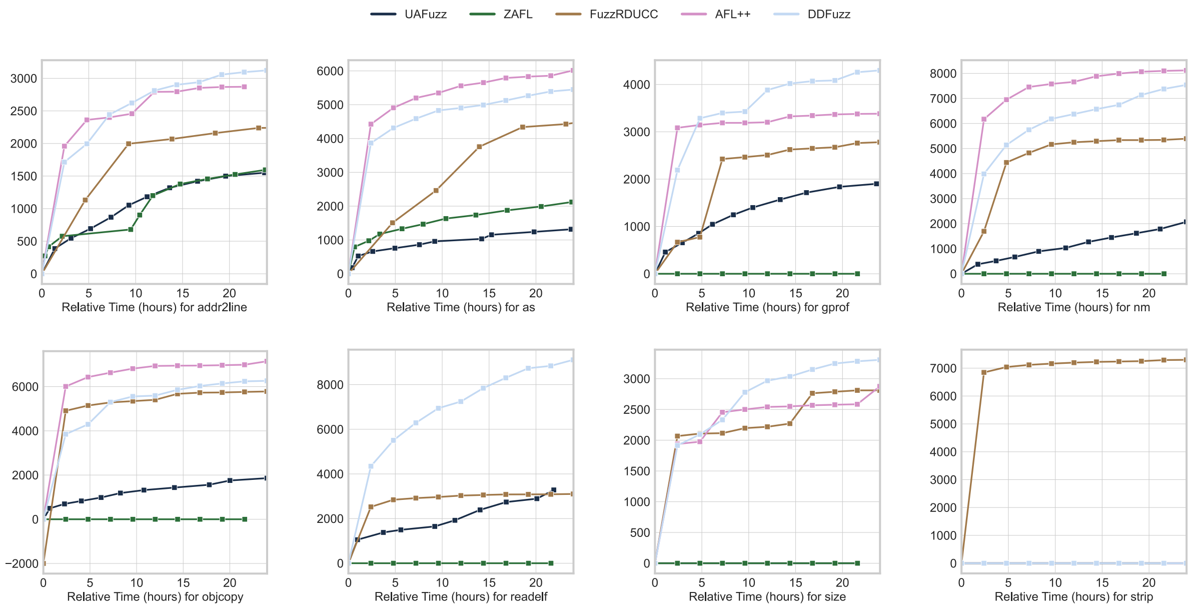


Figure 4.2: Change of Edge Coverage (y-axis) within 24 hours (x-axis) for Fuzzing of Binutils Tools

4.4.1 Evaluation Setup

Baseline: In this study, we compare our proposed framework with several established fuzzing tools, including AFL++, DDfuzz [141], uafuzz [145], and ZAFL [146]. Our analysis focuses on their design principles and operational efficiencies, particularly in relation to dataflow analysis. AFL++ serves as the baseline for fuzzing comparisons, building upon features from AFLfast, Mopt [22], and other advancements in AFL-based tools. DDfuzz introduces a dataflow-driven feedback mechanism that extends beyond control flow edge discovery by guiding fuzzing based on a data dependency graph, although it only supports source code. Our work complements DDfuzz by providing similar dataflow-based feedback for binary only targets. FuzzRDUCC reconstructs def-use chains from stripped executables and instruments QEMU, so it can be applied when source code and recompilation are not available, which is common for firmware and third-party libraries. uafuzz specializes in binary-directed fuzzing to detect use-after-free vulnerabilities, using novel seed metrics to select appropriate seeds for mutation. ZAFL, on the other hand, enhances binary-only fuzzing through binary rewriting to achieve compiler-quality instrumentation. Notably, except for ZAFL, all the mentioned fuzzers utilize the QEMU model, a common framework for emulation-based fuzzing.

Dataset: To comprehensively assess the effectiveness of the fuzzing frameworks, we use a diverse set of binaries, with a focus on GNU Binutils—a widely used suite of binary tools. The selection of Binutils is driven by its critical role within the Linux ecosystem, making it a well-established benchmark for fuzzing evaluations. We test 8 binaries from the Binutils collection to evaluate the fuzzers under consideration.

Evaluation Metric

- **The Change of Edge Coverage**: The effectiveness of a fuzzer is measured by its ability to increase edge coverage, which provides valuable insights into the program's execution paths. Edge coverage acts as a feedback mechanism, helping the fuzzer explore uncharted execution paths and uncover potential vulnerabilities. This metric is pivotal in gauging how thoroughly a fuzzer explores the program's execution space.
- **Number of Crashes**: Crashes are a key indicator of a fuzzer's efficacy, as they signal the discovery of potential vulnerabilities or software defects. A higher number of crashes directly correlates with the fuzzer's ability to uncover significant issues in the target software, making crash detection a vital evaluation criterion.

- **Execution Speed**: When fuzzing binary targets, particularly with QEMU-based fuzzers, there are inevitable performance overheads. It is important to balance between maximizing edge coverage and minimizing performance degradation. Assessing this trade-off is crucial to determine a fuzzer's effectiveness in binary analysis, as maintaining efficiency while reducing overhead is key to successful fuzzing operations.

4.4.2 Preliminary Results

Question 1: Assessing the Feasibility of Implementing Coverage-Based Fuzzing through Dataflow Analysis

To address the first question, we analyse the evolution of edge coverage shown in Figure 4.2. A substantial increase in edge coverage typically correlates with a fuzzer's ability to discover new execution paths. On the majority of the benchmarks, the baseline fuzzers (AFL++ and DDfuzz) demonstrate superior performance: on seven of the eight Binutils tools (excluding `strip`), one or both of these tools achieve the highest coverage. Their curves rise rapidly within the first few hours and stabilise near the top of each plot. In contrast, FuzzRDUCC generally accumulates coverage more slowly than these top-performing baselines, though it consistently outperforms UAFuzz and ZAFL over the 24-hour duration.

This performance hierarchy reflects the inherent trade-off between feedback precision and execution cost. FuzzRDUCC's instrumentation tracks selected def-use chains and updates a secondary bitmap for every translated block. While this yields richer feedback, it imposes a throughput penalty; as illustrated in Figure 4.4, FuzzRDUCC averages approximately 150 executions per second, whereas AFL++ and DDfuzz maintain speeds between 300 and 400 executions per second. Consequently, FuzzRDUCC explores fewer test cases within the fixed time budget.

However, the `strip` benchmark presents a notable exception. For this target, the edge coverage of all baseline fuzzers remains negligible (near zero), whereas FuzzRDUCC rapidly ascends to approximately 7,000 edges and sustains incremental progress. This deviation indicates that dataflow guided feedback is decisive when code exploration requires satisfying structured value flows, such as those found in the relocation handling logic of `strip`, which probabilistic mutations alone fail to penetrate.

Question 2: Benchmarking the Bug Discovery Capabilities of Dataflow Coverage-Based Fuzzing

Figure 4.3 compares the total number of crashes discovered per target. AFL++ proves to be the most prolific bug finder on standard targets, recording the highest crash counts on five of the eight tools (`addr2line`, `as`, `gprof`, `nm`, and `size`). DDfuzz also performs strongly, securing the highest crash counts on `objcopy` and `readelf`. Our prototype, FuzzRDUCC, does not surpass these mature tools in terms of raw bug counts on general targets. This result is consistent with the throughput disparity noted above: with roughly half the execution speed of the baselines, FuzzRDUCC has fewer opportunities to trigger faults. Furthermore, its feedback mechanism favours inputs that exercise specific def-use chains, which may not always align with the shortest path to a crash.

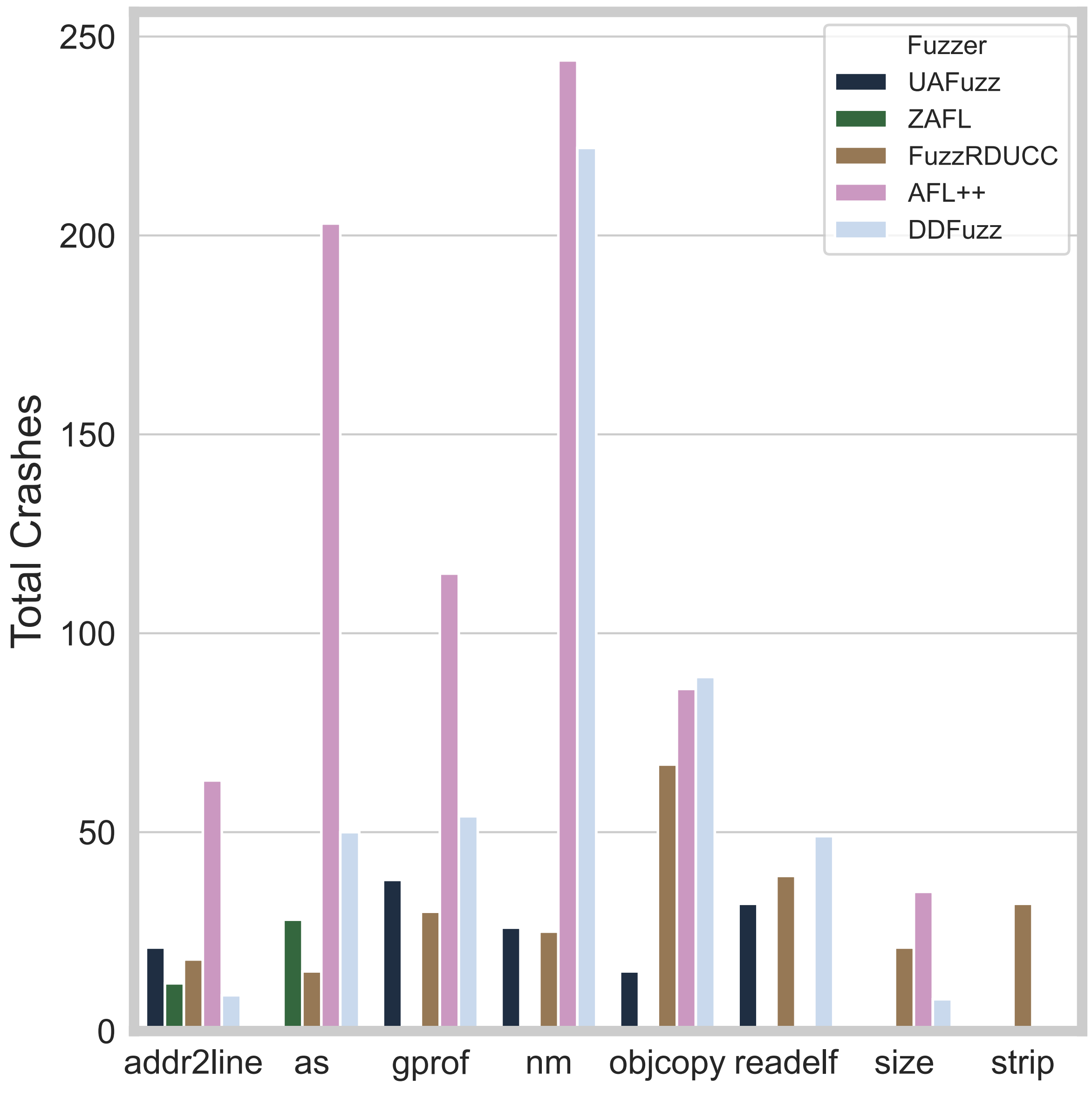


Figure 4.3: Comparison of Crashes Across Different Targets for Each Fuzzer

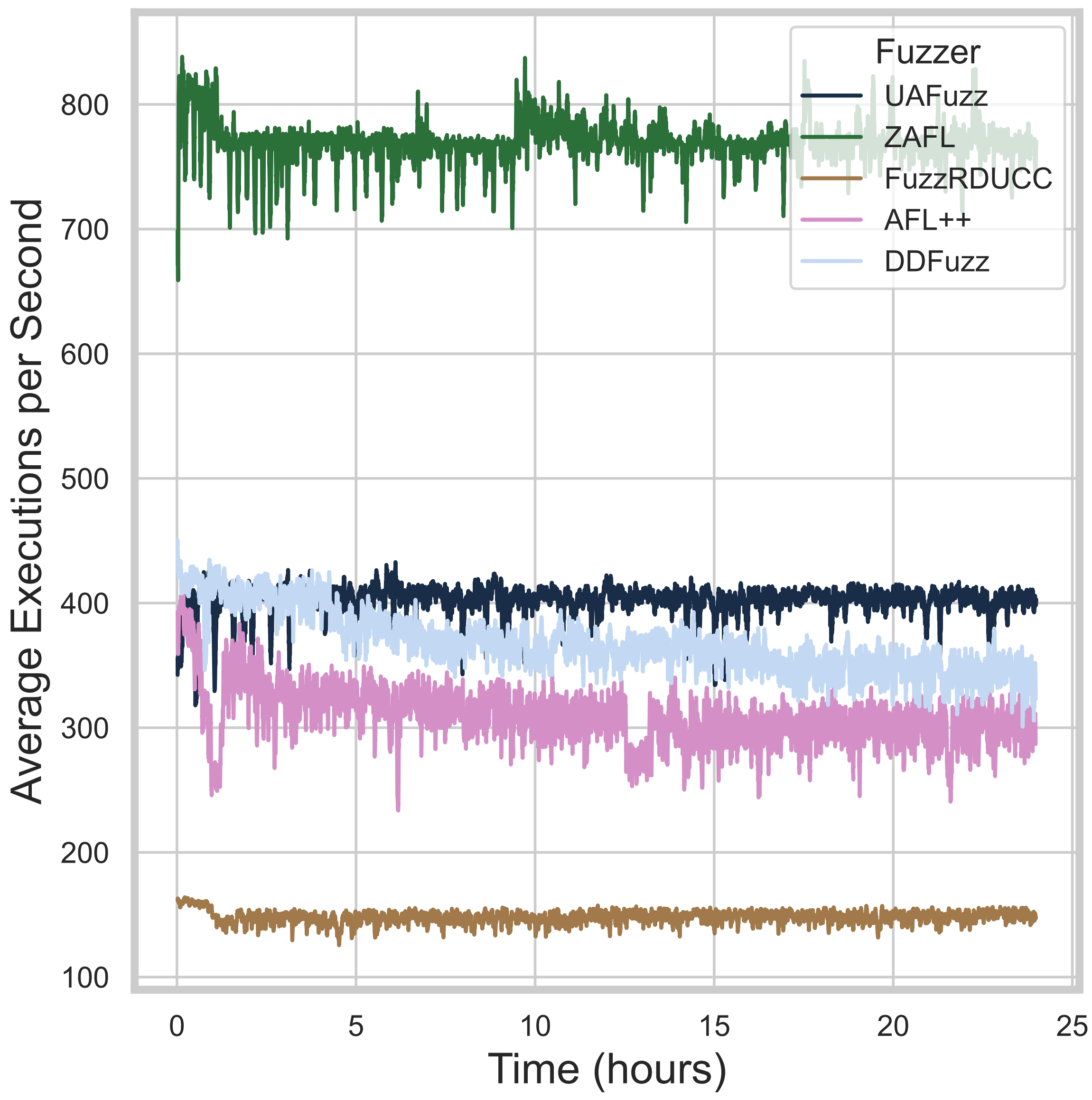


Figure 4.4: Average Execution Speed Over Time for Each Fuzzer

Nevertheless, the results for `strip` again highlight the specific utility of dataflow coverage. FuzzRDUCC is the only fuzzer to discover crashes on this target, while all others report zero. The def-use chain analysis enables the fuzzer to preserve and mutate inputs that successfully reach the relocation-handling code, even if they are initially rejected by earlier validity checks. Specifically, FuzzRDUCC learns to manipulate relocation entries such that a pointer used within `copy_relocations_in_section` becomes invalid. When the program subsequently iterates over these entries, it dereferences the null pointer, triggering a segmentation fault. Fuzzers relying primarily on edge coverage (like AFL++) fail to negotiate the initial validation required to reach this logic, explaining their inability to expose the bug. These findings demonstrate that while binary dataflow coverage may not maximise crash counts on simpler targets, it is capable of uncovering deep, data dependent vulnerabilities that remain invisible to edge based approaches.

Question 3: the Runtime Overhead for Dataflow-Based Fuzzing Compared to Control Flow Fuzzing

Figure 4.4 details the average execution speed over the 24-hour period. ZAFL achieves the highest throughput (approximately 800 executions per second) due to its use of static binary rewriting with lightweight instrumentation. AFL++ and DDfuzz follow, reaching roughly 300-400 executions per second. By comparison, FuzzRDUCC sustains about 150 executions per second. This performance is achieved after applying our def-use selection heuristic; prior to this optimisation, the prototype averaged only 50 executions per second. Despite the three-fold speedup, the overhead of tracking def-use chains in QEMU remains significant. Each executed block incurs costs for def-use table lookups and bitmap updates. This reduced throughput is a primary factor contributing to FuzzRDUCC's lower total edge coverage and crash counts on general targets, as seen in Figures 4.2 and 4.3.

In summary, the results for coverage, crash discovery, and throughput provide a nuanced answer to these questions. FuzzRDUCC demonstrates that dataflow coverage fuzzing for binary is practically viable and capable of revealing faults. However, the experiments also expose clear limitations: the current implementation significantly lags behind AFL++ and DDfuzz in execution speed, which translates to fewer discovered crashes on standard benchmarks.

4.4.3 Future Evaluation

Building on our preliminary evaluation, we plan to conduct the following in-depth experiments:

1. Reducing Overhead via Selective Def-Use Chain Implementation

We aim to significantly reduce binary analysis overhead by selectively implementing def-use chains. By analysing and categorizing each def-use chain using various static analysis algorithms, we will identify the most impactful chains affecting memory changes, striving for a balance between soundness and completeness. We will also simplify def-use chains related to heap memory and optimize the size of translated blocks in QEMU to further decrease emulation overhead.

2. Enhancing Fuzzing Performance with Def-Use Chain Guidance

We hypothesize that def-use chain-guided fuzzing will outperform traditional methods in triggering crashes and detecting vulnerabilities. By focusing fuzzing efforts on specific def-use chains that represent critical paths potentially harbouring vulnerabilities, we aim to increase efficiency and uncover flaws that broader methods may miss. This approach also involves identifying unique vulnerabilities among the detected crashes.

3. Developing Dataflow Coverage Metrics

To better understand the impact of def-use chains, we plan to propose a new metric for dataflow coverage. Since existing fuzzers primarily use bitmaps to track control flow coverage—insufficient for representing dataflow trends—introducing a dataflow coverage metric will help us assess coverage more accurately. This metric will also assist in selecting appropriate hash functions for computing def-use chains, thereby reducing hash collisions.

4. Applying the Framework to Real-World Scenarios

To validate our hypotheses, we are conducting baseline evaluations with GNU Binutils. We will extend our tests to other datasets, such as Magma [27] and Fuzzbench [147], and plan to experiment with real-world IoT firmware using datasets from [6, 81]. These experiments will demonstrate our framework's adaptability and effectiveness across diverse environments.

4.5 Conclusion

We have established the feasibility of using reconstructed def-use chains as feedback to drive the fuzzing process. We have developed a framework designed to recover def-use chains from binary code, thereby providing a new coverage mechanism for grey-box fuzzers. Preliminary results on binutils suggest that our framework successfully identifies some unique crashes, although it incurs relatively high overhead. Future work in chapter 5 will focus on reducing this runtime overhead and conducting more comprehensive evaluations. We also provide source code required to replicate the experiments presented in this chapter.[1]

1. https://github.com/MaksimFeng/AFLplusplus

Chapter 5

Hardfuzz: On-Device Def-Use-Guided Fuzzing with Hardware Breakpoints

5.1 Introduction and Motivation

Research Question 3: *Can on-device fuzzing with hardware breakpoints deliver high-fidelity execution and strong feedback at practical speed for MCUs?*

The first part is to achieve high fidelity execution. Testing on Microcontroller Units (MCUs) presents unique difficulties, although firmware is often compact in terms of lines of code, the primary challenges stem from the complex interaction between the code and its hardware environment rather than code size alone. First, MCU programs are tightly coupled to sensors and actuators through memory-mapped I/O. Inputs arrive as GPIO levels, ADC readings, timer events, or messages on serial buses, and firmware frequently polls or reacts to these signals at precise intervals. To exercise these execution paths, a test harness must either control the physical peripherals or emulate them with high fidelity, both of which are labour intensive and fragile tasks.

Second, MCUs operate under strict resource constraints. The limited RAM and flash memory, small stack sizes, and absence of a full operating system make it difficult to deploy heavy instrumentation or monitoring tools. Many standard testing techniques assume process isolation, virtual memory, or rich debugging interfaces, none of which are typically available on microcontrollers. Finally, concurrency and timing play a central role: interrupts, DMA transfers, and low-power modes interact with application code in ways that are difficult to reproduce deterministically. Small variations in interrupt timing can lead to divergent control flows, complicating both test design and the interpretation of results.

Rehosting techniques, such as full-system emulation, para-rehosting, and hardware-in-the-loop (HiT) approaches, have been proposed to run firmware in controlled environments. specifically designed for firmware analysis. However, achieving high fidelity execution remains a significant hurdle. Rehosting involves decoupling firmware from its underlying hardware to enable execution in a virtual environment; yet, this process is highly complex as firmware is typically compiled for a specific System on Chip (SoC) and interacts with a fixed set of peripherals. It is often impossible to perfectly model all hardware components, leading to semantic discrepancies between the emulated environment and the actual hardware. Furthermore, performance overhead remains a major drawback, with rehosting in QEMU incurring slowdowns of up to 1300% [20]. In contrast, fully on-device execution offers an alternative by utilizing hardware tracing mechanisms, such as Intel Processor Trace (PT) and ARM Embedded Trace Macrocell (ETM), to capture execution flow. However, these methods are not universally applicable, as many embedded devices lack the necessary built-in tracing capabilities, thereby limiting their adoption in practical scenarios.

The second part is to achieve rich feedback in fuzzing process, particularly for MCUs. Traditional coverage-guided fuzzing relies on software instrumentation to monitor code execution. However, this approach is not suitable for the resource-constrained nature of MCUs [87, 148]. The instrumentation overhead increases firmware size, often exceeding the limited memory available. It also slows down execution, which reduces the overall efficiency and throughput of the fuzzing process.

Furthermore, the feedback from control-flow coverage, which is typically gathered at the basic block or edge level, lacks the granularity needed to guide a fuzzer toward deep or complex vulnerabilities. For example, in the function shown in Listing 5.1, a fuzzer guided by CFG coverage can easily generate inputs to pass the initial boundary checks, such as `if (n < 8)`. However, it struggles with the subsequent checks that depend on specific computed values, like if `((((token >> 8) + len) % 29u) != 7u)`. Once the basic blocks for these checks are covered, the CFG-guided fuzzer receives no further guidance. It cannot distinguish between an input that produces a result of 6 and one that produces 20, even though the former is much closer to the target value of 7. The fuzzer has no information about which part of the input—the header bytes `in[0]` and `in[1]` or the payload—is responsible for the values of `token` and `len`. Consequently, a CFG-based fuzzer must rely on random mutations to solve these conditions and may expend a vast amount of time without ever reaching the vulnerable memcpy operation.

In contrast, data-flow-based fuzzing not only tracks which paths are executed but also monitors how data values are defined and used throughout the program. This approach provides a much richer feedback mechanism. For instance, in Listing 5.1, a data-flow fuzzer can identify that the definition of `len` (D1) is directly used in the `memcpy` function (U3) and that the definition of `token` (D2) is used in two conditional checks (U1 and U2). By tracking these definition-use (def-use) pairs, the fuzzer gains granular insight, allowing it to correlate specific input bytes with their effects on program state and more effectively navigate complex conditional logic.

```
static uint32_t crc32_like(const uint8_t *p, size_t n) {
    uint32_t h = 0x811C9DC5u;
    for (size_t i = 0; i < n; ++i) { h ^= p[i]; h *= 16777619u; }
        return h;
}

int process_packet(const uint8_t in, size_t n) {
    if (n < 8) return -1;
    uint16_t len = (uint16_t)((in[0] << 8) | in[1]);         //
    if (len > n - 4) return -2;                   //simple bound check
    const uint8_t payload = in + 4;
    uint32_t token = crc32_like(payload, len) ^ 0x5A5A5A5AA; //(D2)
    if (((token ^ 0xA5A5A5A5A) & 0x3u) != 0)                 //(U1)
        return 0;
    if ((((token >> 8) + len) % 29u) != 7u)                  //(U2)
        return 0;
    uint8_t buf[128];
    memcpy(buf, payload, len);                               //(U3)
    return 1;
}
```

Listing 5.1: Example where def-use guidance provides earlier signals than basic-block coverage. Marks (D1,D2) are definitions; (U1,U2,U3) are uses.

Listing 5.1 exposes three def-use pairs:

- **(D1 $\rightarrow$ U3)**: the definition of `len` is used as the length in `memcpy`.
- **(D2 $\rightarrow$ U1)**: the definition of `token` is used in the first gate.

Case	**CFG coverage signal**	**DU coverage signal**
Input reaches U1 only	none	new pair (D2→U1)
Input reaches U1 and U2	none	new pairs (D2→U1), (D2→U2)
Input reaches U3	new block	new pair (D1→U3)

Table 5.1: Feedback per input category. DU guidance provides earlier, better signals.

- **(D2 → U2)**: the same `token` is used again in the second gate.

A control flow based fuzzer records new blocks. After it first reaches the blocks that implement U1 and U2, reaching them again with different values provides no new control flow signal. Inputs that almost satisfy the gates (e.g., reaching U1 but narrowly failing the condition) are not rewarded and are often discarded. The deep path behind both U1 and U2, along with the large `len` required at U3, may remain undiscovered because the fuzzer receives no intermediate gain in CFG coverage to guide its mutations.

Def-use chain guidance rewards value flows rather than just new control-flow blocks. When execution reaches U1 and reads the value defined at D2, the fuzzer records the pair (D2→U1), even if the branch itself is not taken. This provides an intermediate signal, encouraging the fuzzer to retain the input in its corpus and to concentrate mutations on the bytes that influenced this dataflow. Later, when execution reaches U2, the fuzzer records the additional pair (D2→U2). Finally, when the program reaches the `memcpy`, it records (D1→U3). The distinct feedback generated at each stage is summarized in Table 5.1. These incremental signals form a gradient of progress, guiding the fuzzer more effectively toward the deep path compared with control-flow coverage alone.

In practice, these intermediate def-use signals help preserve and improve the right seeds —those that set `token` and `len` to values that are closer to satisfying the required conditions. As a result, a fuzzer with def-use guidance reaches the deep path in significantly fewer iterations than one guided by control flow coverage alone.

To address this research question **RQ3**, achieving high-fidelity execution and strong feedback, we propose Hardfuzz, an on-device fuzzing framework that guides exploration using definition-use (def-use) chains in the program. A def-use chain links a program point where a variable is defined with subsequent points that use that value. By targeting def-use chains, Hardfuzz goes beyond basic-block coverage to drive the fuzzer toward inputs that not only reach new code locations but also cause specific data-flow interactions to occur.

Hardfuzz operates directly on the device under test (DUT), using the debug unit's limited hardware breakpoint registers to catch executions of selected def-use chain points. It integrates with a feedback-driven input generator, monitoring def-hit and def-use hit events as coverage signals. The overall goal is to discover subtle states in the program (e.g., a sequence of variable assignments and uses leading to a bug) that pure control-flow coverage might miss.

5.2 Hardfuzz Overview

Hardfuzz combines an offline static analysis stage with an online fuzzing loop to systematically cover def-use chains on an embedded device. Figure 5.1 illustrates the overall architecture of Hardfuzz, which can be divided into three main phases: (1) Static Analysis & setup, (2) Def-use-guided fuzz loop, and (3) Coverage-driven input generation.

1. Static Analysis & Setup: Before fuzzing, we analyse the target program's binary to extract all def-use chains. This yields a set of definition addresses each paired with one or more use addresses. The Hardfuzz runner loads this information and initializes its components: the GDB controller, serial connection, metrics logger, and input generator. The GDB controller attaches to the device or emulator and performs an initial reset/halt, inserting a breakpoint at main and running to that

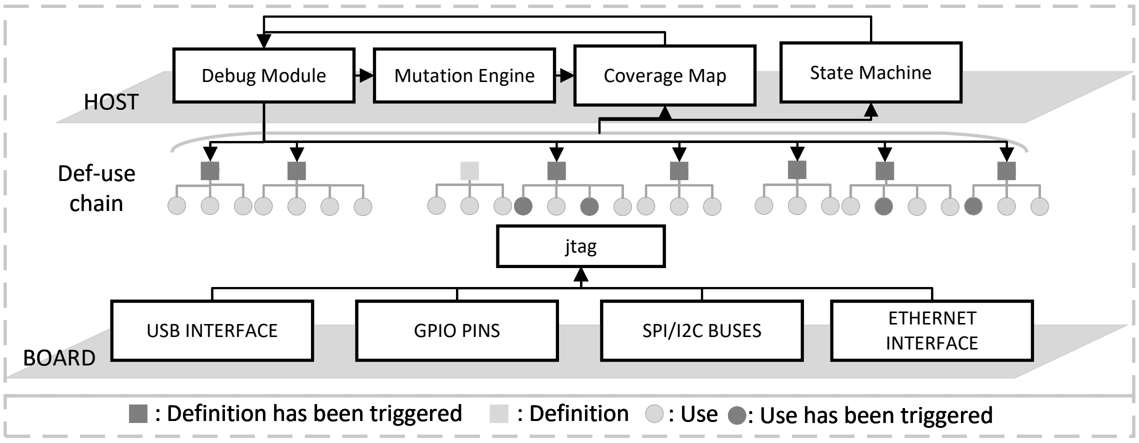


Figure 5.1: Hardfuzz Overview

point. The serial connection thread is started to handle input/output with the target. The input generator is seeded either with user-provided seed inputs or a default seed; it maintains the corpus of interesting inputs discovered. At this stage, Hardfuzz also precomputes some helper structures from the def-use list, such as a mapping of each def address to the basic block containing it (and likewise for uses). It also computes a weighting for each def (for fuzzing schedule) based on the number of uses it has.

2. Def-Use-Guided Fuzz Loop: Hardfuzz then enters the main fuzzing loop, which runs infinite rounds of test generation and execution. In each round, an input is selected and mutated, then used to execute a series of def-use chain trials. Unlike a pure coverage fuzzer that would run one input and simply note which new blocks were hit, Hardfuzz actively guides each input run towards a specific def-use target. It works as follows: it selects a subset of def addresses (up to the hardware breakpoint limit, e.g. 6) that have not yet been fully covered, and sets hardware breakpoints at those definition addresses (marked as Def-BPs). Then it releases the target to run the test input from the beginning. If none of those definitions executes (no breakpoint hit), the input did not trigger those targets; Hardfuzz will then try a different set of def addresses (or a new input in the next round). If one of the def breakpoints hits, the execution stops at that definition point. At this moment, Hardfuzz identifies which def was hit and retrieves its list of corresponding use addresses. It then immediately arms a second set of breakpoints for those uses

(marking them Use-BPs) and resumes execution. The original def breakpoint, being temporary, is auto-removed upon hit to free a slot. Now the target continues running the same input, but with breakpoints set at the uses of the just-hit definition. If any of those uses executes, the program will halt again at the use site (indicating the def-use chain was successfully realized at runtime). Hardfuzz logs this as a def-use pair covered and removes the use breakpoint. It allows the program to continue, potentially catching multiple uses in one execution if the input triggers more than one use of the definition's value. Once the program completes (or a timeout/crash occurs), Hardfuzz cleans up any remaining breakpoints and resets the target if needed before the next round.

3. Coverage-Driven Input Generation: After each test execution, Hardfuzz updates its coverage bitmap to reflect any newly covered def or def-use pair. It uses two 64kB bitmaps in shared memory: one for def coverage (indexed by def address bits) and one for def-use coverage (indexed by a hash of def and use addresses). Any time a definition is hit or a def-use pair is completed, the corresponding bits are set. At the end of a round, Hardfuzz checks if any new bits were set compared to the global ”virgin” coverage map. If new coverage was found, the input that achieved it is saved to the corpus and considered for fuzzing again in the future. The fuzzer then chooses a new baseline input for mutation-it may choose the latest high-value input or cycle through the corpus to keep diversity. Hardfuzz employs a mutation engine based on libFuzzer's mutator: by linking against the LLVM libFuzzer mutation library, it can generate mutated variants of an input efficiently. In each round, one or more new candidate inputs are produced this way. If a round produced no new coverage (no def-use hit and no new def hit), Hardfuzz can retry with a different def target or eventually switch to a fresh mutated input. This coverage-driven strategy ensures that Hardfuzz concentrates on inputs that expand the def-use coverage frontier.

5.3 Def-Use Chain Analysis and Selection

5.3.1 Def-Use Chain Analysis

We extract *definition-use (def-use) chains* from MCU's firmware binaries to guide test generation and breakpoint placement. A *definition site* (def) is an instruction that writes a program value (a register or a memory location). A *use site* (use) is an instruction that reads that value. A *def-use chain* is a directed edge from a def instruction to a use instruction along some feasible path in the data dependence graph (DDG). We use these chains to (i) measure dataflow coverage and (ii) prioritize fuzzing inputs that reach definitions with many uses.

Our analysis runs in three phases. First, We load the target ELF with `angr` and build a context-sensitive data dependence graph (DDG). For each discovered function, we run `ReachingDefinitions` based on angr's intermediate represent (VEX IR) to compute the set of definitions that may reach each program point. For each definition we found, we enumerate its uses with instruction address and check for reachability in the DDG and CFG. If there exists a path from the def to the use, we add an edge $a_d \rightarrow a_u$ to the def-use graph. The graph also contains chains that cross function boundaries (e.g., def in caller, use in callee). The details show in Algorithm 3.

5.3.2 Breakpoint Strategy

After extracting the def-use chains, we prioritize definitions to guide the fuzzer's exploration. The goal is to focus on definitions that influence many uses, as they are more likely to lead to diverse program behaviours and potential vulnerabilities. We also consider the history of selections to avoid over-focusing on a few definitions. We assign each definition

Algorithm 3: Def-Use Chain Extraction (per function)

```
Input: Function $F$, Data Dependence Graph $G$
Output: Set $\mathscr{E}$ of pairs $(def\_addr, use\_addr)$
1  $\mathscr{E} \leftarrow \varnothing$
2  RD $\leftarrow$ ReachingDefinitions($F$)
3  foreach $d \in$ RD.$all\_definitions$ do
4      $n_d \leftarrow$ Node($G, d.ins\_addr$)
5      if $n_d = \perp$ then continue
6      foreach $u \in$ GetUses(RD, $d$) do
7          $n_u \leftarrow$ Node($G, u.ins\_addr$)
8          if $n_u = \perp$ then continue
9          if Reachable($G, n_d, n_u$) then
10             $\mathscr{E} \leftarrow \mathscr{E} \cup \{(d.ins\_addr, u.ins\_addr)\}$
11 return $\mathscr{E}$
```

a base weight equal to the minimum the number of distinct uses it has, so definitions with many uses are considered more ”interesting” by default. During fuzzing process, we also adjust weights based on how often a def has been tried locally in the current round and globally across all rounds. Intuitively, if a particular def has already been hit several times (globally) or if we have attempted it repeatedly in the current round, its probability is reduced to avoid too much repetition. The exact formula is described below.

For a definition address a_d with use set $U(a_d)$, the scheduler samples with

$$w(a_d) = \underbrace{\max\bigl(1,\ |U(a_d)|\bigr)}_{\text{base weight}} \cdot \underbrace{\frac{1}{1+\ell(a_d)}}_{\text{local penalty}} \cdot \underbrace{\frac{1}{\bigl(1+g(a_d)\bigr)^{1/2}}}_{\text{global penalty}}.$$

where $\ell(a_d)$ is the *local* count of selections of a_d in the current generator and $g(a_d)$ is the *global* hit/selection count accumulated across rounds. Definitions are drawn by roulette-wheel sampling proportional to $w(a_d)$.

Once a def a_d triggers, we order its uses by address proximity and enable up to K hardware breakpoints (with $K = 6$ on ARM Cortex-M3):

$$\mathrm{order}_U(a_d) \;=\; \underset{a_u \in U(a_d)}{\mathrm{argsort}} \left| a_u - a_d \right|, \qquad S(a_d) \;=\; \text{first } K \text{ of } \mathrm{order}_U(a_d).$$

The intuition is that uses close to the def are more likely to be executed soon after the def, increasing the chance of hitting a use in the same run. If a def has more than K uses, we will not be able to cover them all in one execution. However, since we sample defs multiple times across rounds, we will eventually cover all uses over time. Once one breakpoint hits, we will consider the basic block containing it as covered and remove the breakpoint to free a slot for the next use breakpoint. In this way, we can potentially catch multiple defs in one execution if the input triggers one basic block.

In each fuzzing round, hardfuzz draws up to n definition targets from this weighted generator (with N set to the hardware breakpoint limit) to form a batch. The reason for batching is efficiency: setting breakpoints is slow, and it is wasteful to run on input per breakpoint if we can enable multiple breakpoints at once. Batching also allows one input to potentially cover multiple defs if they happen to be hit in the same execution. The batch is constructed and all breakpoints for that batch are inserted before running the test input. If none of breakpoints in the batch are hit by the time the input finished, it implies the input does not execute any of those defs. In that case, Hardfuzz will fetch the next batch of defs (if any remain untried for this input) and rerun the same input on a fresh instance of the program. This approach gives each input multiple opportunities to demonstrate coverage on different def targets. If an input completely fails to hit any new def after exhausting all batches, Hardfuzz will conclude that the input is ”stuck” coverage-wise and move to the next input. In our implementation, we set a limit `(e.g., NO_TRIGGER_-THRESHOLD=8)` on consecutive attempts with no new hits before abandoning an input to avoid infinite loops.

Algorithm 4: Hardware Breakpoint Strategy (Def→Use under comparator budget K)

```
Input: Test input $x$; batch of definitions DefsBatch $\subseteq \mathscr{D}$; use map $U(\cdot)$; HW
       breakpoint limit $K$
Output: HitDef $\in \mathscr{D} \cup \{$None$\}$; set HitPairs $\subseteq \{(d,u)\}$
1  Primitives: HaltThenDeleteAll(), SetHWBP($a$, temporary), ContinueAndFeed($x$),
     WaitStop(), RemoveBP($a$), RestartIfCrashedOrTimedOut()
2  HitDef $\leftarrow$ None, HitPairs $\leftarrow \varnothing$
3  HaltThenDeleteAll()
4  foreach $d \in$ DefsBatch (distinct), up to $K$ do      // arm up to $K$ defs as
     hardware & temporary BPs
5      SetHWBP($d$, temporary = True)
6  ContinueAndFeed($x$)
7  (reason, payload) $\leftarrow$ WaitStop()
8  if reason is "breakpoint hit" and payload is a def BP then
9      HitDef $\leftarrow d^\star$
10 else if reason $\in$ {"timed out", "crashed", "exited"} then
11     RestartIfCrashedOrTimedOut()
12     return (HitDef, HitPairs)
13 HaltThenDeleteAll()                // clean breakpoint before use phase
14 if HitDef = None then
15     return (HitDef, HitPairs)
16 // Use phase: sweep uses of $d^\star$ in chunks of size $K$
17 UsesSorted $\leftarrow$ uses in $U(d^\star)$ sorted by $|u - d^\star|$ (ascending)
18 while untried uses remain do
19     take next chunk UChunk of $\leq K$ addresses from UsesSorted
20     HaltThenDeleteAll()
21     foreach $u \in$ UChunk do              // arm hardware BPs for uses
22         SetHWBP($u$, temporary = False)
23     ContinueAndFeed($x$)
24     (reason, payload) $\leftarrow$ WaitStop()
25     if reason is "breakpoint hit" and payload is a use BP then
26         let $u^\star$ be the hit use
27         HitPairs $\leftarrow$ HitPairs $\cup \{(d^\star, u^\star)\}$
28         RemoveBP($u^\star$)      // free comparator; others remain armed
29         continue
30     else if reason $\in$ {"timed out", "crashed", "exited"} then
31         RestartIfCrashedOrTimedOut(); break
32     else
33         continue                  // no use hit; move to next chunk
34 HaltThenDeleteAll()
35 return (HitDef, HitPairs)
```

The workflow of the breakpoint strategy is summarized in Algorithm 4. Managing the limited hardware breakpoints is a core part of Hardfuzz's design. We implement a lightweight GDB controller that communicates with the target device via GDB's machine interface (MI). The controller provides primitives to set and remove breakpoints, continue execution, wait for stops, and handle crashes or timeouts. These primitives are used in the breakpoint strategy to orchestrate the def-use guided execution.

When a def breakpoint triggers, the GDB stop reason comes as ”breakpoint-hit” with an associated breakpoint number. We determine whether this was one of our def breakpoints by looking it up in the batch mapping. If so, we record the hit and prepare to switch to use breakpoints. To be noticed is that on Arm Cortex-M: when a breakpoint hits at an instruction in flash, the processor actually replaces that instruction with a BKPT instruction internally. If we immediately removed the breakpoint and continued, we risk re-executing the BKPT instead of the original instruction. To avoid this, Hardfuzz performs a single-step operation to execute the instruction and move past it before inserting new breakpoints. This ensures the def instruction completes and the PC advances, preventing any ”flash breakpoint deadlock” where the same breakpoint would re-trigger or corrupt execution. Our *BreakpointManager* handles this: upon detecting a def breakpoint number, it executes one instruction step, then clears all existing use breakpoints from any previous def, and finally removes the def breakpoint itself to free the slot. After that, Hardfuzz proceeds to install the use breakpoints for the triggered def.

After each batch (or after a def-use sequence completes), Hardfuzz issues a blanket `-break-delete` command to clear any leftover breakpoints before moving on. This is important to prevent stray breakpoints from persisting into the next input's execution, which could cause false coverage signals or unintended halts. We found that after heavy churn of breakpoints, it was sometimes necessary to stabilize the GDB connection. In

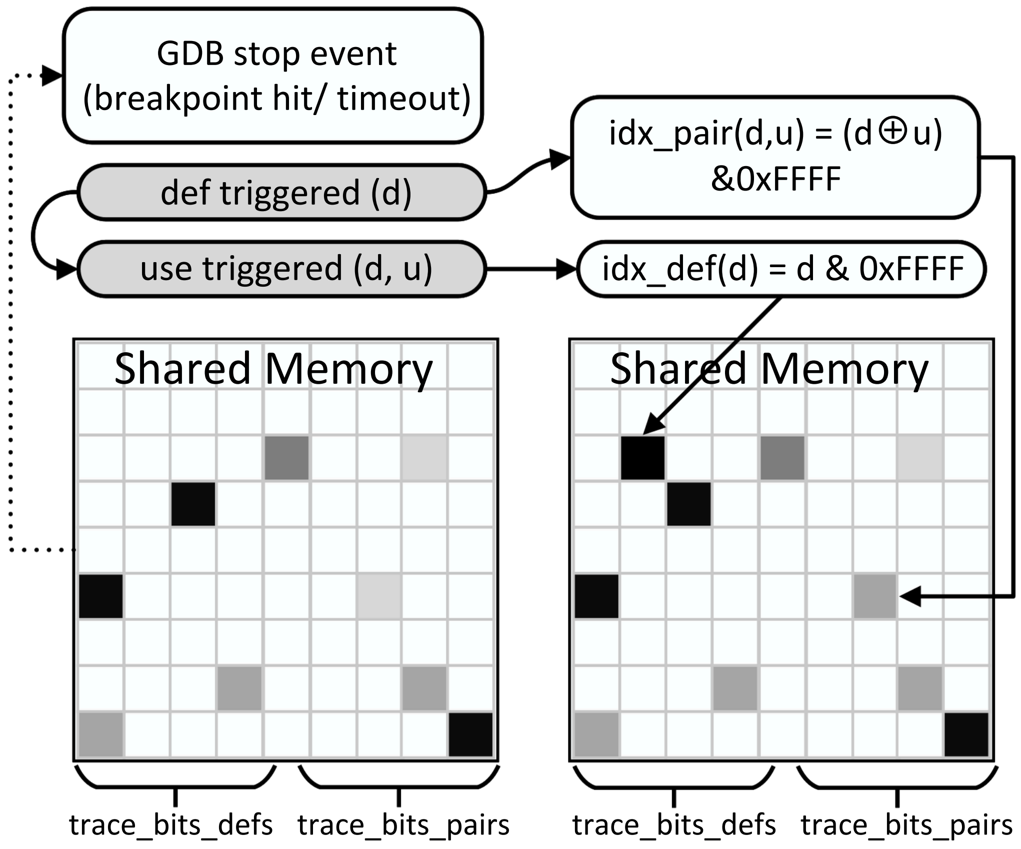


Figure 5.2: Two-bitmaps in shared memory and update flow. A breakpoint hit yields d (and optionally u). The CoverageManager computes `idx_def`$(d) = d$ & $0x$FFFF and `idx_pair`$(d,u) = (d \oplus u)$ & $0x$FFFF, then sets the corresponding bits in the two bitmaps (NumPy views backed by one shared-memory region of size $2M$). Darkness indicates the time for the triggers to activate. Virgin maps flip from $0x$FF to $0x$00 on first observation and gate corpus updates.

extreme cases (e.g., if the target becomes unresponsive or GDB misbehaves), Hardfuzz will restart the GDB session by killing the old GDB and launching a new one, then re-attaching to the target. This ”GDB rejuvenation” is triggered after certain timeouts or errors to maintain a robust fuzzing run.

5.3.3 Coverage Guidance

Hardfuzz needs a light-weight signal that can run on the device, without binary rewriting, and that still shows progress on data flow. We therefore record two events: (i) a definition is executed; and (ii) a definition-use pair is executed. We turn these events into coverage using two compact bitmaps stored in one shared-memory block (see Figure 5.2).

We allocate a single shared-memory region of size $2M$ bytes and split it into two non-overlapping slices:

$$\texttt{trace\_bits\_defs}[0..M-1] \quad \text{and} \quad \texttt{trace\_bits\_pairs}[M..2M-1].$$

In our implementation $M = 65{,}536$. Each slice is a byte array used as a bitmap (0 or 1 per slot). This design lets the fuzzer and the coverage code communicate without copying and keeps the memory footprint fixed.

We map events to indices as follows.

- **Definition coverage.** When a def at address d executes,

  $$\texttt{idx\_def}(d) = d \,\&\, 0x\mathrm{FFFF},$$

  and we set $\texttt{trace\_bits\_defs}[\texttt{idx\_def}(d)] \leftarrow 1$.
- **Def-use coverage.** When a use at address u executes *after* the matching def at d in the same input run,

  $$\texttt{idx\_pair}(d,u) = (d \oplus u) \,\&\, 0x\mathrm{FFFF},$$

  and we set $\texttt{trace\_bits\_pairs}[\texttt{idx\_pair}(d,u)] \leftarrow 1$.

The XOR gives a constant-time hash from a pair of addresses to one slot. Collisions can happen but are rare at this scale; they may reduce granularity but do not break the guidance.

The figure shows three states of the same shared-memory block.

Before: both bitmaps reflect the current round before the new stop event.$\rightarrow$ *After a def hit*: one cell in `trace_bits_defs` $[0,M)$ is set to 1 $\rightarrow$ *After a def then use hit*: one cell in `trace_bits_pairs` $[M,2M)$ is also set to 1 $\rightarrow$ Placing both slices inside the same box and labelling $[0,M)$ and $[M,2M)$ makes clear that the bitmaps share memory but do not overlap.

If the execution passes through a basic block without stopping inside it, we conservatively mark: (i) every def located in that block as covered; and (ii) every (d,u) pair whose use lies in that block as covered. We do this using a precomputed lookup from each block to its defs and to the (d,u) pairs whose u is in that block. This avoids setting a breakpoint at every use site while still rewarding progress once the block executes.

To decide if an input should be kept, we maintain two ”virgin” arrays in process memory, `fresh_defs`$[0..M-1]$ and `fresh_pairs`$[0..M-1]$, initialized to $0x\mathrm{FF}$. After running an input we scan the two shared bitmaps. For each index k:

$$\texttt{trace\_bits\_defs}[k] \neq 0 \ \wedge\ \texttt{fresh\_defs}[k] = 0x\mathrm{FF} \ \Rightarrow\ \texttt{fresh\_defs}[k] \leftarrow 0x00,$$
$$\texttt{trace\_bits\_pairs}[k] \neq 0 \ \wedge\ \texttt{fresh\_pairs}[k] = 0x\mathrm{FF} \ \Rightarrow\ \texttt{fresh\_pairs}[k] \leftarrow 0x00.$$

If at least one byte flips from $0x\mathrm{FF}$ to $0x00$, the input exposed new coverage. We then add the input to the corpus and optionally pick it (or a mutated child) as the next baseline. The shared bitmaps are cleared for the next input, while the virgin arrays keep the lifetime view of what has already been discovered.

Basic-block coverage rewards only new control flow. Our two-bitmap scheme adds a data-flow signal. The def bitmap rewards reaching a definition; the pair bitmap rewards reaching a use of that definition. These intermediate signals give the fuzzer a gradient toward the deep path even when no new basic block is covered.

5.4 Evaluation

Research Question 3: *Can on-device fuzzing with hardware breakpoints deliver high-fidelity execution and strong feedback at practical speed for MCUs?*

We therefore compare Hardfuzz against GDBFuzz in two settings: (i) QEMU-based emulation, where both fuzzers run on the same emulated targets, and (ii) on device execution on real microcontrollers. In both cases we measure coverage growth and basic blocks reached to understand when the breakpoint based def-use feedback improves over baseline.

5.4.1 Experimental Setup

We evaluated Hardfuzz against GDBFuzz on two platforms: (1) an emulated environment using QEMU to simulate an ARM Cortex-M3 firmware, and (2) a real hardware setup using an Arduino Due board (SAM3X8E MCU) connected via a J-Link debug probe. The fuzzing campaigns were run for a fixed time budget on each platform. For GDBFuzz, which does not natively track def-use chains, we consider only basic block coverage for comparison. All experiments used the same initial seed corpus and were allocated identical time for fairness.

5.4.2 QEMU-Based Emulation Results

In the QEMU emulation, both fuzzers can execute inputs relatively quickly (no physical device latency). In this way, we could compare the two different breakpoint assignment strategies (Hardfuzz's def-use guided vs. GDBFuzz's Dominator-based) under same conditions. We ran each fuzzer for 24 hours in this environment for three repetitions. The target programs we choose are from Google Fuzzbench [147], a well-known benchmark suite for fuzzing research. We selected 16 targets that are compatible with QEMU and also been tested in original GDBFuzz [40]. The results are shown in Figure 5.3.

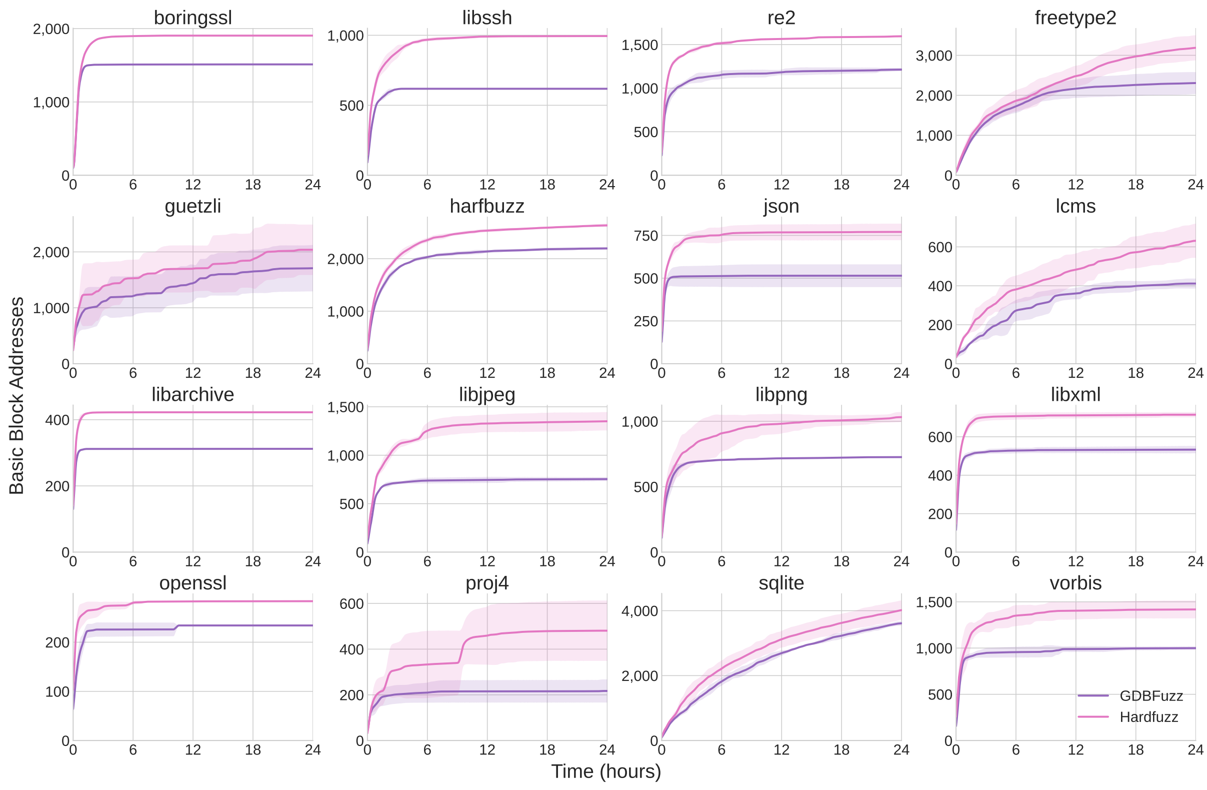


Figure 5.3: QEMU Emulation Results: Basic block coverage achieved by Hardfuzz and GDBFuzz over 24 hours across 16 targets. Hardfuzz consistently outperforms GDBFuzz in most cases, demonstrating the effectiveness of def-use chain guidance in improving coverage.

Figure 5.3 plots basic block coverage over time for the sixteen Magma binaries under QEMU. In all targets the Hardfuzz (pink) curve lies above the GDBFuzz (purple) curve after the first few hours, and the gap either remains stable or widens over the 24 hour campaign. For example, in `freetype2` and `lcms` Hardfuzz continues to discover new blocks throughout the whole run, while GDBFuzz reaches a clear plateau after roughly 6-8 hours. The continued upward trend for Hardfuzz indicates that def-use guidance keeps proposing inputs that exercise additional data-flow chains, whereas the dominator based strategy in GDBFuzz quickly exhausts easy to reach control flow edges and then spends most of its time revisiting already covered regions.

Across graphics and parsing heavy libraries such as `guetzli`, `harfbuzz`, `libpng`, and `libxml`, Hardfuzz attains noticeably higher final coverage, suggesting that tracking how values are defined and later used is particularly helpful in code with long computations and layered transformations. In contrast, for simpler utilities such as `boringssl`, `libssh`, and `re2`, both fuzzers saturate quickly and the distance between the curves is smaller: once the relatively shallow control flow has been explored, there are fewer hard to reach def-use chains for Hardfuzz to exploit. There are also programs where GDBFuzz comes close to Hardfuzz at the 24 hour mark (for example `freetype2` and `sqlite`), which suggests that in some code bases the dominator-based heuristic happens to align reasonably well with the underlying data flow and therefore narrows the advantage of explicit def-use guidance. Overall, however, the aggregated trend over all sixteen binaries is that Hardfuzz both reaches higher coverage and maintains non-zero coverage growth for longer, supporting the claim that dataflow feedback provides a richer exploration signal than control flow structure alone.

The unique basic block results in bar Figure 5.4 further reinforce Hardfuzz's advantage. Over the 24-hour period, Hardfuzz consistently discovers more unique blocks than GDBFuzz, indicating that its def-use chain guidance effectively drives exploration into new areas of the codebase. The GDBFuzz can achieve the similar results in only two targets (freetype2 and sqlite). This suggests that while dominator-based selection can be effective in certain scenarios, it generally lacks the nuanced direction provided by def-use analysis.

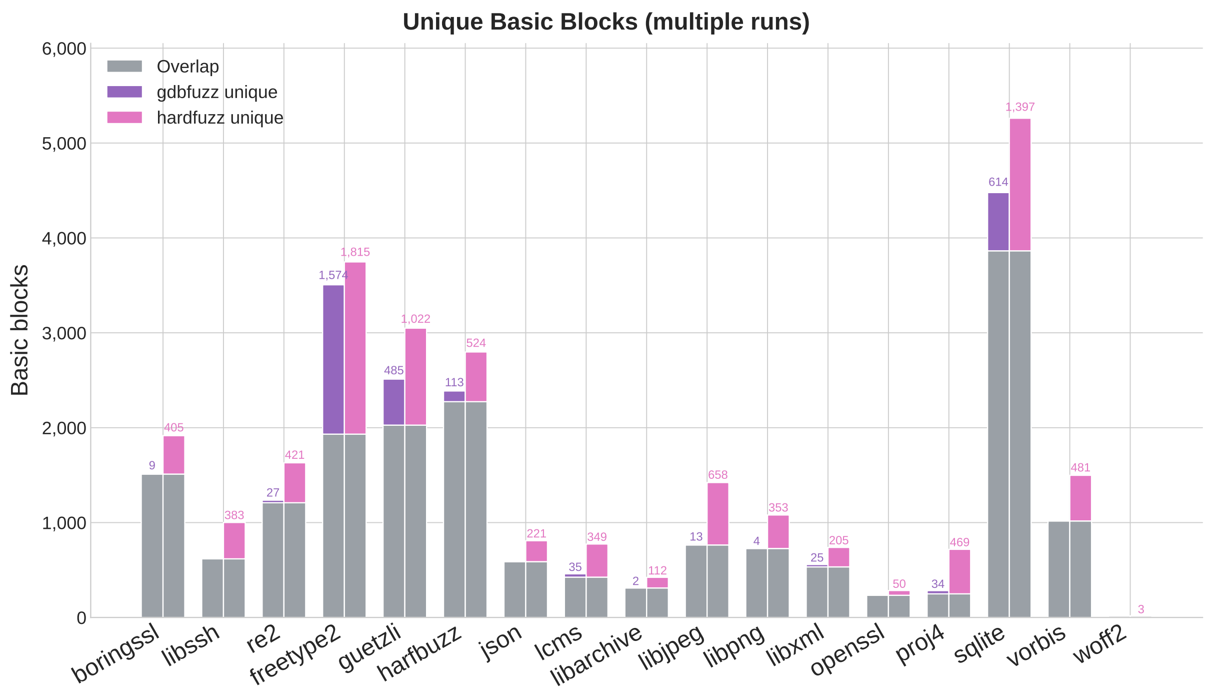


Figure 5.4: Unique basic block coverage over time on QEMU. Hardfuzz consistently discovers more unique blocks than GDBFuzz, demonstrating its superior exploration capabilities.

The ability to target specific data-flow interactions allows Hardfuzz to uncover paths that may be overlooked when focusing solely on control-flow structures. The results highlight the importance of considering both control and data flow in fuzzing strategies to maximize coverage and discovery potential.

5.4.3 On-Device Hardware Results

We also evaluated Hardfuzz on a real device: an Arduino Due (SAM3X8E) connected through a J-Link. Running on physical hardware adds latency from the debug link and the lower clock speed of the MCU, but it gives us ground-truth signals (hardware faults and precise stop points). We ran both Hardfuzz and GDBFuzz for 24 hours of three repetitions on this setup. The firmware targets are the same types used in our GDBFuzz experiments, and each contains a small, known bug so we can measure detection and deduplication. The three targets are:

Table 5.2: Basic block coverage on hardware after 24 hours

Target	Basic Blocks Covered	
	GDBFuzz	Hardfuzz
buggycode	62/249	88/249
HTTP server	373/1504	524/1504
JSON parser	664/1071	758/1071

1. **buggycode (stack overflow).** A minimal UART harness that looks for the four-byte gate `"bug!"` and then copies the received payload into a fixed 20-byte stack buffer without bounds checks. Any input longer than 20 bytes triggers a deterministic overflow.
2. **HTTP server (state-machine bug).** A small ESP-IDF HTTP service with an endpoint that mixes fixed-length responses with chunked sends in the same request. This violates the servers send path and produces a reproducible failure under load, modelling common handler mistakes in embedded web servers.
3. **JSON parser (length-triggered hang).** A serial JSON parser built with ArduinoJson that reads a 32-bit length prefix. If the length exceeds the configured buffer size, the firmware enters a persistent wait state. This gives us a clean timeout class distinct from crashes.

Table 5.2 presents the final basic block coverage achieved on hardware after a 24 hour campaign. Hardfuzz outperforms GDBFuzz across all three firmware targets. On the `buggycode` harness, coverage increases from 62 to 88 blocks. A similar relative improvement is observed for the HTTP server, where coverage rises from 373 to 524 blocks. This suggests that Hardfuzz is more effective at navigating the state machine and exercising diverse request-handling paths. The JSON parser exhibits a smaller but distinct gain, improving from 664 to 758 blocks. These results demonstrate that def-use guidance successfully exposes additional behaviours even in small codebases like `buggycode`, with benefits that become increasingly evident as control logic complexity grows, as seen in the HTTP server.

Figure 5.5 provides the mean coverage over three repetitions with shaded regions representing one standard deviation. For `buggycode`, both fuzzers saturate quickly, reaching a plateau within the first hour. While Hardfuzz converges to a higher final value than GDBFuzz, the standard deviation bands overlap significantly.

In contrast, the HTTP server displays a more distinct pattern. Hardfuzz accelerates coverage discovery early in the campaign and maintains a substantial lead throughout the 24 hour period. The separation between the mean curves is pronounced, with only marginal overlap in the uncertainty bands, indicating that this performance advantage is systematic rather than the result of stochastic variance in a single run. For the JSON parser, Hardfuzz consistently maintains a lead over GDBFuzz.

Ideally, differences in fuzzer performance, such as final coverage or the area under the curve, should be validated using non-parametric two-sample tests (e.g., the Mann-Whitney U test), as recommended by standard evaluation guidelines [14]. However, given the resource constraints of hardware in the loop testing, our dataset is limited to three repetitions per configuration. With such a small sample size, statistical tests lack sufficient power and yield unstable p values. Consequently, we prefer descriptive analysis rather than formal hypothesis testing; we rely on mean trajectories and variance bands to assess the magnitude and consistency of the observed differences. Future work involving larger scale campaigns could supplement this qualitative assessment with rigorous statistical validation.

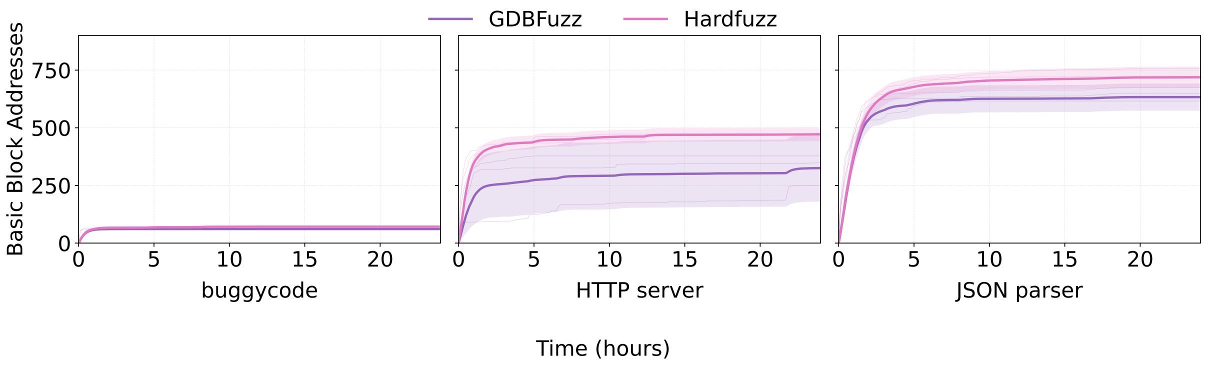


Figure 5.5: Coverage changes over time on hardware. Hardfuzz consistently discovers more unique blocks than GDBFuzz, demonstrating its superior exploration capabilities.

To be noticed is that Hardfuzz's higher coverage does not come from extra online analysis. We extract def-use chains with angr offline before fuzzing, so this step adds no runtime cost. By contrast, GDBFuzz updates control-flow information during fuzzing when it finds new coverage, which adds some overhead. Both systems rely on GDB stop reasons for crash detection; halts, breakpoint churn, and occasional re-attach process also cost time on real hardware. Despite these costs, Hardfuzz's richer signals yield better exploration and explain the observed coverage gains.

5.5 Limitations and Future Work

Hardfuzz improves fidelity and feedback on MCUs, but it does not replace emulation or rehosting. These approaches are complementary. Emulation scales and is easy to automate across many targets. On-device fuzzing gives ground truth behaviour but pays for I/O latency and debug overheads and is tied to specific boards.

Cortex-M parts expose only a small number of hardware comparators (e.g., six on Cortex-M3 for breakpoints). Arming and rearming breakpoints through GDB adds latency, and some devices lack trace mechanisms entirely. This limits the number of def/use pairs we can watch at once and caps per second executions. We plan to combine flash breakpoint with watchpoints, use RAM software breakpoints, amortize re-programming with persistent execution loops on the target, and opportunistically use trace (ETM/ITM/ETB) or RTT mailboxes when available to cut halt/resume cycles [149].

Semihosting, SWD/JTAG, and serial handshakes add delay. Our current design halts to set breakpoints and to step past breakpoints, which reduces cycles per second. Future work could design a persistent harness that processes many testcases per boot, a small on-target control loop to arm next breakpoints via a memory-mapped index table without global halt.

We build def-use chains from binaries. Optimized builds, inlining and register allocation can blur the mapping from IR to concrete addresses and drop some uses. Stripped binaries reduce function recovery quality. The future work includes: (1) fall back to dynamic analysis ;(2) add a lightweight dynamic taint or value-flow sampler to validate and refine static pairs; (3) consume optional symbols or minimal debug info when present; and (4) model common library idioms to cut false pairs.

As shown in Figure 5.3 and Figure 5.4, the code coverage growth eventually slows down and plateaus. This happens because both fuzzers use libfuzzer's mutation strategy, which relies on a random combination of simple changes like bit flips, byte flips, and arithmetic operations. While this method is effective for exploring a broad range of inputs initially, it often struggles to generate the specific inputs needed to bypass complex checks and reach deep program states. As a result, the fuzzer reaches a point of saturation, after which finding new code paths becomes rare, and coverage growth is negligible. Therefore, instead of running the fuzzer for a fixed, long duration like 24 hours, it is more practical to analyse this saturation point to determine an efficient time budget for the fuzzing campaign [150, 151].

On hardware we used three focused targets with known faults; in emulation we used a larger corpus. This is useful for controlled comparisons, but broader external validity needs more firmware, more boards, and blind bugs. Future work contains scale to community firmware, report time-to-first-crash and deduped bug counts.

5.6 Conclusion

This chapter proposes Hardfuzz, a on-device dataflow guided fuzzer for embedded systems. Hardfuzz use hardware breakpoints to monitor the change of the def-use chains, providing precise and efficient feedback to guide the fuzzing process. The evaluation results show that Hardfuzz outperforms the state-of-the-art GDBFuzz in both emulated and real hardware environments, achieving higher code coverage and discovering more unique basic blocks. This demonstrates the effectiveness of def-use chain guidance in improving the exploration capabilities of fuzzers for embedded systems. The source code of Hardfuzz is available online[1] for further research and development in this area.

1. https://github.com/MaksimFeng/Hardfuzz

Chapter 6

Differential testing of MicroPython under CHERI

6.1 Introduction and Motivation

Research Question 4: *As firmware complexity grows, traditional fuzzing struggles with highly structured inputs like language interpreters. How can we evolve test generation beyond simple mutation to rigorously assess architectural defences, thereby measuring the shift from vulnerability discovery to prevention?*

This chapter addresses RQ4 by exploring the application of differential testing to evaluate the security benefits of architectural memory safety mechanisms, specifically CHERI, in the context of embedded language interpreters. As embedded systems increasingly incorporate complex software components like interpreters, they become more susceptible to memory safety vulnerabilities. Traditional fuzzing techniques often fall short in effectively testing such structured inputs, necessitating more sophisticated approaches.

In this chapter our main contribution is a differential testing framework that uses a CHERI-enabled interpreter as a hardware backed test oracle for memory safety. The key idea is to execute the same, syntactically valid MicroPython program on two binaries that share the same source code but differ in their hardware protection model: a conventional baseline and a CHERI build. Whenever the baseline terminates normally or crashes while the CHERI build raises a capability violation, we treat the CHERI outcome as the reference behaviour of a memory safe execution and flag the input as exposing a bug in the unprotected configuration.

Microcontroller firmware frequently embeds a high-level interpreter to speed up development and to keep device firmware small and flexible. MicroPython is a widely used Python 3 implementation for resource-constrained systems. Fuzzing such interpreters is attractive but difficult: naive mutation produces many syntactically invalid programs and often fails to expose the memory hazards that matter most in the interpreter's underlying C core. At the same time, new hardware architectures like CHERI can enforce bounds, permissions, and provenance on pointers at runtime, turning latent memory errors into precise user-space traps.

Concretely, we instantiate this framework on MicroPython as follows. We build an initial seed corpus from public bug issues and CVE reports for CPython and MicroPython, and then use a prompt guided large language model to synthesise additional well formed programs that resemble these seeds. A concrete syntax tree (CST) mutator based on LibCST[1] applies structure preserving edits (such as inserting statements, wrapping expressions, or changing literal values) to expand the corpus while keeping programs parsable and type-correct. A dual-lane harness executes each test under identical resource limits on both the non-CHERI and CHERI builds and records normal outputs, exceptions, signals, and capability violations. The differential comparator then acts as the oracle: it reports a

1. https://libcst.readthedocs.io/en/latest/index.html

potential memory safety bug whenever the non-CHERI run crashes, hangs, or produces an inconsistent result, while the CHERI run stops with a precise capability fault. The following sections detail the technical background, architecture, and evaluation of this CHERI-based oracle.

6.2 Technical Background

6.2.1 The Architectural Foundation of CHERI Memory Protection

Capability Hardware Enhanced RISC Instructions (CHERI) is an ISA extension that augments conventional processors with *architectural capabilities* to deliver fine-grained memory safety and scalable compartmentalization. Rather than adding checks around unsafe code, CHERI changes the fundamental contract between software and memory by replacing raw integer pointers with hardware-enforced, unforgeable *capabilities*. This targets the root causes of memory-safety bugs that have long dominated systems software (roughly 70% of modern security issues are memory errors) [152]. As we will show, this design is the driver of the divergent behaviours observed under differential testing [153].

A CHERI capability is an atomic token that combines: (i) an integer address, (ii) metadata describing authority, and (iii) a 1-bit validity tag. For example, on a 64-bit architecture a conventional 64-bit pointer is replaced by a token holding a 64-bit address, *circa* 64-bit metadata, and a 1-bit tag, shown in figure 6.1. The metadata encodes:

- **Bounds** The range of the authorized buffer;
- **Architectural permissions (AP)** Operations such as read, write, execute, and capability load/store.

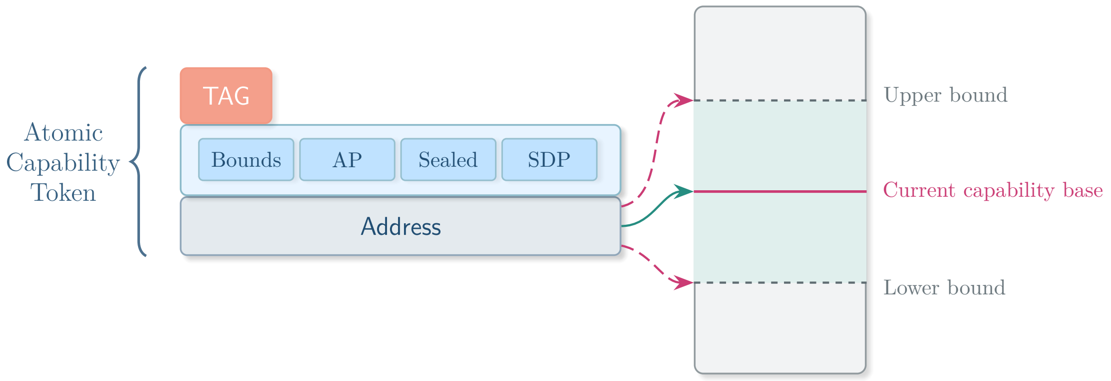


Figure 6.1: CHERI capability token and address-space view. The token combines an integer Address with metadata slices for Bounds, Architectural Permissions (AP), Sealed, and Software-Defined Permissions (SDP), plus a 1-bit TAG that records validity. The shaded region marks the in-bounds range; the solid green arrow indicates a permitted access, while dashed red arrows indicate out-of-bounds accesses that trap.

- **A sealing bit** Making a capability immutable and non-dereferenceable until explicitly unsealed;
- **Software-defined permissions (SDP)** Configuration of implementation or OS specific policy.

The tag records capability integrity. Tags are maintained by hardware: software can read them but cannot set them directly. Using an untagged capability (e.g., one corrupted by non-capability writes) triggers a hardware exception.

CHERI enforces three key properties on capabilities:

- *Provenance*: Every capability must be derived from an existing valid capability through sanctioned operations;
- *Monotonicity*: Derivation can only reduce or preserve rights or bounds, authority cannot increase;
- *Integrity*: Forged or corrupted capabilities cannot be dereferenced.

Sealing further strengthens control flow and encapsulation: sealed capabilities are immutable and non-dereferenceable until unsealed with the matching authority. For example, CHERI designs employ sealed entry capabilities for return addresses and function pointers so that code capabilities cannot be freely modified or misused.

At reset, the program counter capability (PCC) and default data capability (DDC) begin with wide bounds and broad permissions. The boot loader and OS then reduce these authorities: the OS receives a memory capability and tightens PCC/DDC; user processes are created with derived, restricted capabilities for their address spaces and objects. This staged restriction embodies provenance and monotonicity: all application capabilities descend from a small set of boot-time roots and become strictly less powerful over time.

CHERI reifies pointer correctness in hardware. Three common classes of misuse trigger precise traps [154]:

Bounds violation: Access falls outside the capability's lower/upper bounds.

Permission violation: An operation (e.g., write or execute) is not permitted by the capability's permission bits.

Tag violation: The operation attempts to use an untagged capability.

On a CHERI system these conditions fault at the exact offending instruction, preventing memory corruption from ever occurring. On a conventional system, the same bug might silently corrupt state and only crash much later (or not at all). This difference produces early, unambiguous signals in our experiments: CHERI typically traps deterministically at the point of error, while the baseline may exhibit delayed or non-deterministic failures. This is precisely the divergence our differential testing is designed to surface [153].

6.2.2 Differential Testing

Differential testing (also known as back-to-back testing) is a software testing technique designed to detect semantic or logical bugs by providing identical inputs to two or more different implementations of the same specification and comparing their outputs [155]. Its primary strength lies in uncovering discrepancies that do not necessarily lead to obvious failures like crashes or assertion failures [156]. In many testing scenarios, defining a single correct output for a complex input can be prohibitively difficult-a challenge known as the test oracle problem [157]. Differential testing sidesteps this issue by designing oracles for each other. The fundamental assumption is that while implementations may differ internally, they should produce externally equivalent behaviour for the same inputs [158].

In the context of our work, the term ”output” is defined broadly to include not only a program’s standard output stream, but also its exit status, error messages printed to standard error, signals received from the operating system, and even whether a core dump file is produced. Any divergence in this comprehensive set of observable behaviours between the CHERI and non-CHERI executions is treated as a significant result, pointing directly to architectural differences between the two platforms.

Our use of differential testing for MicroPython on CHERI adapts the methodology for a unique purpose. The non-CHERI system serves as the baseline, which is a control group representing the standard, insecure behaviour of a C-based interpreter on conventional hardware. The behaviours observed on this baseline (including crashes, silent data corruptions, and other unpredictable outcomes) are essentially the ”expected” results in a world without hardware-enforced memory safety. The CHERI-enabled system, running the same MicroPython interpreter compiled for the CHERI ABI, is the new implementation under test.

This framing fundamentally shifts the objective of the differential comparison. In a typical differential test, any discrepancy between two implementations would indicate a bug in at least one of them. In our scenario, however, divergent behaviour is intentional and desired. The CHERI architecture is explicitly designed to behave differently when confronted with a memory safety violation. A test case that causes silent memory corruption on the non-CHERI platform is expected to trigger a deterministic hardware trap on the CHERI platform. Therefore, the primary goal of our differential tests is not to find bugs in the MicroPython interpreter itself, but rather to empirically verify, characterize, and demonstrate the security advantages of the CHERI architecture. For example, if the CHERI build produces a `SIGPROT` (protection violation signal) where the non-CHERI build produces a `SIGSEGV` (segmentation fault) or, worse, completes execution while silently corrupting memory, that outcome is considered a successful validation of CHERI's safety guarantees. In this way, the experiment is transformed from a bug-finding exercise into a scientific validation of a new security paradigm, using the conventional system as a baseline illustration of the very problems CHERI is intended to solve. In operational terms, the CHERI-enabled build therefore serves as our test oracle: whenever its execution diverges from the baseline in the form of a precise capability fault, we interpret the CHERI outcome as the correct memory-safe behaviour and judge the conventional build against it.

6.2.3 MicroPython

The choice of MicroPython as the software under test is particularly strategic [159]. As a high-level, dynamically-typed language, Python is designed to be memory-safe from the programmer's perspective. However, the MicroPython interpreter, which executes the Python code, is itself a complex program written primarily in C. This creates a ideal test case: a widely-used piece of software whose high-level safety guarantees depend entirely on the low-level memory integrity of its C implementation. Subjecting this interpreter to differential testing provides a clear view into how CHERI's protections can harden critical runtime systems.

MicroPython consists of a compiler that translates Python source code into bytecode and a virtual machine that interprets and executes that bytecode. This entire toolchain-the parser, compiler, object system, garbage collector, and the implementations of all built-in functions and modules-is written in C. Consequently, despite the memory safety of the Python language itself, the interpreter is vulnerable to the full spectrum of memory errors endemic to C programming, including buffer overflows, use-after-free vulnerabilities, and invalid pointer manipulations.

The MicroPython has been migrated to CHERI [160], making it an excellent candidate for our differential testing framework. The interpreter's complexity and its reliance on C for core functionality mean that it is likely to contain latent memory safety issues that CHERI can help mitigate. By running the same Python scripts on both the CHERI-enabled and non-CHERI builds of MicroPython, we can directly observe how CHERI's architectural features influence the interpreter's behaviour in the presence of memory errors. This setup allows us to not only identify potential vulnerabilities in the interpreter but also to demonstrate how CHERI can transform these vulnerabilities from silent, exploitable flaws into well-defined, manageable exceptions [161].

At the same time, MicroPython should be viewed as a representative example rather than the ultimate target of our contribution. Our goal is not to design MicroPython-specific defences, but to evaluate how a CHERI-enabled build can act as a general oracle for memory safety in C-based interpreters and runtimes. In principle, any similar interpreter that can be compiled for CHERI could be dropped into the same differential-testing harness with minimal changes, the core methodology and oracle logic would remain the same. For instance, the recent CHERI port of CRuby could be directly integrated into our differential testing framework [153].

6.3 Differential Testing Overview

Figure 6.2 provides a high-level overview of our differential testing framework, which is organized into three layers. The first layer produces a corpus of valid programs. It starts from curated scripts aligned with known CVEs in CPython and MicroPython and public bug reports. Then, a test-case generator (guided by prompts encoding patterns known to be risky in MicroPython) synthesizes new seed programs that remain within the subset of Python features supported by our target platform. A mutator built on LibCST next applies structure-preserving transformations to broaden the input space. Each candidate input is validated, de-duplicated, and added to a unified corpus that feeds both execution lanes.

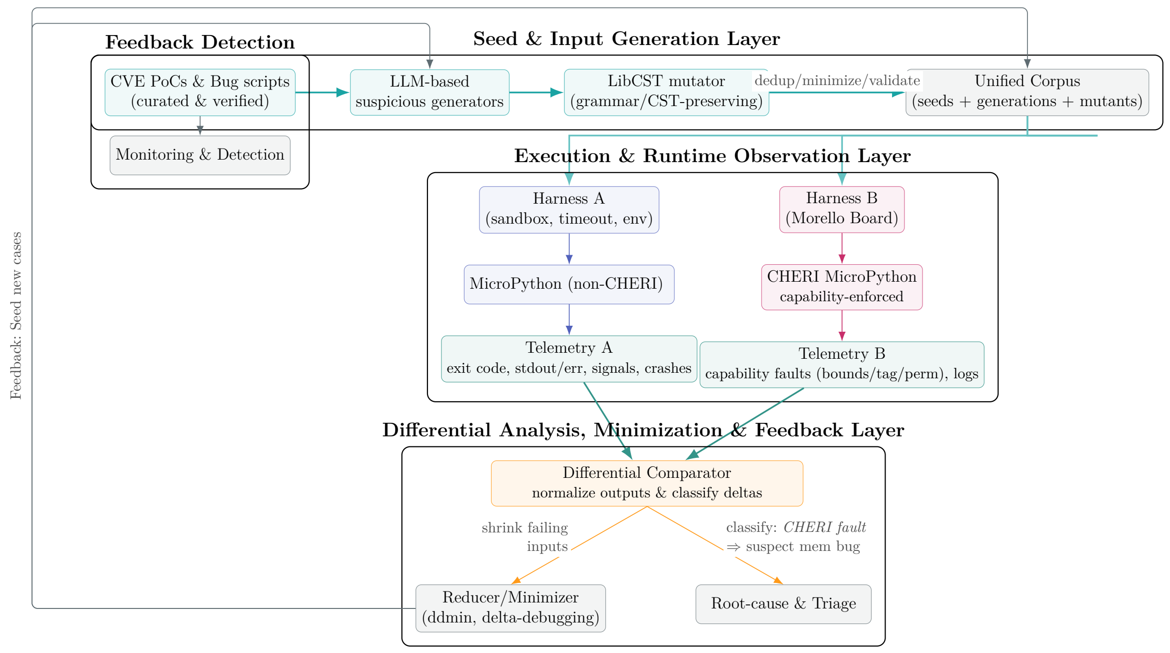

Feedback Detection
Seed & Input Generation Layer
CVE PoCs & Bug scripts (curated & verified)
LLM-based suspicious generators
LibCST mutator (grammar/CST-preserving)
dedup/minimize/validate
Unified Corpus (seeds + generations + mutants)
Monitoring & Detection
Execution & Runtime Observation Layer
Harness A (sandbox, timeout, env)
Harness B (Morello Board)
MicroPython (non-CHERI)
CHERI MicroPython capability-enforced
Telemetry A exit code, stdout/err, signals, crashes
Telemetry B capability faults (bounds/tag/perm), logs
Feedback: Seed new cases
Differential Analysis, Minimization & Feedback Layer
Differential Comparator normalize outputs & classify deltas
shrink failing inputs
classify: CHERI fault ⇒ suspect mem bug
Reducer/Minimizer (ddmin, delta-debugging)
Root-cause & Triage

Figure 6.2: Differential testing framework for MicroPython with CHERI.

The second layer runs each test program on two builds of MicroPython. The left lane uses a normal (non-CHERI) build of MicroPython, executed inside a sandbox on a conventional system with strict time and memory limits. The right lane uses a CHERI-enabled build running on hardware that supports capability enforcement, the ARM Morello prototype board. The harness on each side captures the program's output, exit code, and any signals or crashes; on the CHERI side it also logs any capability fault details (such as bounds, tag, or permission violations). To enable a fair comparison, the harness normalizes non-deterministic aspects of outputs (e.g. memory addresses in error logs or ephemeral file names in tracebacks).

The third layer compares the two runs. It classifies the pair into one of several categories. The category that matters most for memory safety is the one where the baseline crashes or exhibits undefined behaviour while the CHERI build reports a capability violation or exits normally. The comparator assigns a stable signature to each discrepancy using a small set of fields drawn from the termination state and from a normalized summary of the top frames. A reducer then shrinks the input. The result feeds back into the corpus and into the prompt context for the generator so that future runs start from richer seeds.

6.4 Methodology

The implementation of our framework follows the three-layer structure described above. In particular, the methodology is divided into input generation (layer1), execution and monitoring (layer2), and differential analysis plus bug triage (layer3).

Table 6.1: Vulnerability and Bug Report Classification

Category	**Verified Bug Reports**	**CVEs**
Raw memory, buffer protocol & view lifetime	16	19
Binary conversions & bigint corners	4	14
FFI / native emitters	1	5
Parsers, codecs & compressors in C	68	15
Filesystem, VFS & Race Conditions	12	19
MMIO & peripherals (embedded targets)	0	10
Interpreter internals, exceptions & GC	38	18
Total	**139**	**100**

6.4.1 Input Generation and Corpus Management

Our approach to generating test inputs consists of three stages: **(A)** seeding the process with a collection of real-world bug scripts and CVE proofs-of-concept, **(B)** using an LLM-based generator to produce new candidate programs, and **(C)** applying a LibCST-based mutator to systematically introduce variations. We describe each stage in turn.

A: *The collection of CVE PoCs and Bug Reports*

Our methodology begins with a curated corpus of proof-of-concept (PoC) scripts derived from known CVEs and bug reports affecting both CPython and MicroPython. This initial set provides a solid foundation grounded in real-world vulnerabilities. By analysing these reports, we identify common bug patterns and high-risk programming constructs, particularly those relevant to the resource-constrained environment of the MicroPython interpreter. These seed scripts serve as the basis for further generation and mutation, allowing us to explore novel security issues beyond the known vulnerability landscape.

The results of this classification are summarized in Table 6.1. The analysis reveals several key areas of concern. The most prolific category in terms of implementation errors is Parsers, codecs & compressors in C, with 68 verified bug reports. This is unsurprising, as these components must safely handle a wide variety of complex and often untrusted data formats, making them a frequent source of crashes or unexpected behaviur when processing malformed input.

From a security perspective, the Raw memory, buffer protocol & view lifetime category is particularly critical, accounting for 19 CVEs and 16 bug reports. This category covers direct memory manipulation, where objects like memoryview can point to another object's memory (e.g., a bytearray) without copying data. Such a view can become a dangling pointer if the underlying object is modified or deallocated, leading to high-impact vulnerabilities like buffer overflows and use-after-free, which can often be exploited for arbitrary code execution.

Additionally, the Interpreter internals, exceptions & GC category is also significant, with 38 bugs and 18 CVEs. These issues relate to the most complex and interdependent parts of the interpreter. This category tests the core functionality of the interpreter, and vulnerabilities here can destabilize the entire system, leading to crashes or memory leaks.

Other important categories include:

- Binary conversions & bigint corners: This category focuses on vulnerabilities in the conversion between Python integers and byte sequences, which can lead to size calculation errors, integer overflows, and misaligned memory access when handling binary data.
- FFI / native emitters: This category covers vulnerabilities related to the interaction between Python and C libraries, where mismatches in function prototypes, misuse of variable arguments, or unsafe pointer arithmetic can lead to crashes and memory corruption.

- Filesystem, VFS & Race Conditions: This category includes vulnerabilities in file I/O operations, such as length mismatches in buffer operations, use-after-close errors, and race conditions that can lead to data corruption or security bypasses.
- MMIO & peripherals (embedded targets): This category is specific to embedded systems and deals with vulnerabilities related to direct hardware access, where arbitrary memory access or misalignment can cause system instability or crashes.

B:*LLM-Based Generators*

The initial seed corpus is expanded using two distinct techniques: **(B)** an LLM-based generator and **(C)** a LibCST-based mutator. The LLM-based generator produces new test cases that respect Python syntax and semantics while exploring edge cases and complex constructs that may trigger vulnerabilities in the MicroPython interpreter. We do not perform any additional model training, instead, we rely on in-context *prompting*, which leverages the model's prior knowledge of Python acquired during pre-training.

Specifically, we guide generation by providing an instruction prefix to OpenAI GPT-5. This prefix includes:

1. a concise description of MicroPython and its execution model;
2. a compilation of risky code patterns observed in the seed corpus; and
3. selected CVE proofs of concept and bug-report scripts that exemplify these patterns.

The prompt further instructs the model to assign each generated test to one of the categories. The category information follow the pattern in Table 6.1. it also imposes constraints to ensure the outputs remain within the subset of Python features supported by MicroPython. This process excludes constructs that are either known to be safe or deemed irrelevant for our research objectives. The prompt was iteratively refined based on the quality and relevance of the generated scripts, a process intended to maximize diversity and potential impact. Complete prompt templates are provided in Appendix A.

C:*LibCST Mutator*

After generating new testcases using the LLM, we further expand our corpus through designing a *context-aware, type-aware Python code mutator* built on LibCST. Library of Concrete Syntax Tree (LibCST) is a Python parsing and rewriting toolkit that preserves formatting details such as whitespace and comments, ensuring that mutated code remains syntactically valid and stylistically consistent with the original source. We define a serial of mutation rules to introduce subtle variations into the programs while respecting contextual constraints (e.g. scope and syntax) and optional type constraints [162]. The goal is to generate a diverse set of program variants for testing without breaking syntax or introducing glaring type inconsistencies.

Key design goals of the mutator include:

(i) **Preserve syntactic validity and formatting.** By leveraging LibCST's CST, all transformations maintain valid Python syntax and preserve layout, comments, and formatting of unmodified parts of the code. This prevents trivial syntax errors and keeps mutations semantically readable.

(ii) **Context-aware replacements.** The mutator uses scope-sensitive context pools of CST nodes to guide replacements. Any code fragment selected for replacement is substituted with another fragment of a compatible category (expression, statement, etc.), drawn either from elsewhere in the program or from a template library. This ensures that replacements respect the surrounding context (for example, an expression is only replaced with another valid expression of an appropriate type or structure).

(iii) **Multi-pass mutation pipeline.** Instead of applying a single mutation in isolation, the mutator supports executing multiple passes of different mutation operator families (structural, peephole, chaotic, etc.) sequentially in one run. This multi-pass approach allows both coarse-grained structural changes and fine-grained tweaks to be introduced, increasing the chance of complex interactions. Undesired mutation steps can be rolled back or skipped (with type-check gating, as described below) to maintain overall correctness.

(iv) **Type-budget enforcement.** To keep mutations semantically plausible, an optional type-checking gate (using MyPy[2]) enforces a ”type budget.” Before any mutations, we record the original program’s type errors as a baseline. Each candidate mutated program is only accepted if it does not increase the number of type errors beyond that baseline. This prevents the mutator from introducing obvious type violations (such as arity mismatches in function calls or incompatible assignments), thereby preserving a baseline level of semantic consistency.

(v) **Runtime anomaly scoring.** The mutator can optionally execute the final mutated program under MicroPython interpreter with timeout, in order to detect crashes, assertion failures, or infinite loops (hangs). A scoring mechanism assigns higher scores to mutations that trigger abnormal behaviour (non-zero exit status, crashes, timeouts), which is useful for prioritizing interesting or bug-inducing mutants in a fuzzing campaign.

(vi) **Deterministic, reproducible generation.** All randomness in the mutator is driven by a single master seed. Given the same seed and configuration, the mutator will produce the same sequence of mutations every time. This determinism greatly aids debugging and evaluation by ensuring that experiments are repeatable.

The mutation workflow is summarized in Algorithm 5. At a high level, the mutator takes an input source code S and a configuration `cfg` specifying the mutation parameters (such as which operator to apply, how many mutations, whether to enforce type safety, etc.). It then proceeds in four main stages:

2. `https://github.com/python/mypy`

(i) *Parsing and indexing:* The source S is parsed into a LibCST CST with metadata, and a `ContextIndex` is built. The ContextIndex collects nodes from the CST categorized by syntactic type (expressions, statements, control-flow blocks like `if`/`for`/`while`/`try`, function and `class` definitions, etc.), and records scoping information (using LibCST's `ScopeProvider` metadata). This indexing provides pools of candidate nodes available for context-aware replacements or insertions. For example, all expression nodes in the tree might form a pool from which a new random expression can be drawn to replace some target expression, if such an operator is applied.

(ii) *Pipeline assembly:* Based on the specified mutation operation and profile, a sequence of transformation passes Π is constructed (BuildPipeline in Algorithm 5). Each pass corresponds to a certain family of mutation operators.

(iii) *Multi-pass mutation with type gating:* The mutator iterates through each pass $p \in \Pi$ and applies it to the current version of the code $S^\star$ (initially $S^\star = S$). Within a pass, the transformer will select one or more target nodes (guided by the context index and the mutation strategy) and apply the specific mutation operator to produce a modified CST M (Algorithm 5, line 6). For example, a `StructurePass` in replace mode will randomly choose a statement in the code and replace it with another statement drawn from the context pool or a generative template. After each pass produces a candidate M, the Accept function checks if the mutated code should be accepted or rolled back. If type-checking is enabled (`cfg.type_safe` is true), then Accept runs MyPy on M and compares the number of errors to the baseline B. If the candidate M introduces new type errors beyond the allowed budget, it is rejected and the mutation is rolled back (the original $S^\star$ is retained for the next pass). This ensures that type-safe mode yields a series of transformations that, at each step, do not accumulate type inconsistencies. If the candidate is accepted (or if type gating is off), $S^\star$ is updated to M and the next pass continues from this new state. This design, combined with a retry mechanism, allows the mutator to attempt multiple alternatives if a particular mutation site leads to a type error, thereby increasing the chance of finding a valid mutation.

Algorithm 5: Context- and Type-Aware Mutator

```
Input: Source $S$, configuration cfg
Output: Mutated source $S^\star$
1  $T, W, I \leftarrow$ ParseAndIndex($S$)  // LibCST + metadata; build ContextIndex
2  $\Pi \leftarrow$ BuildPipeline(cfg)  // e.g., Swap/Chaos/Structure/Peephole/Noise
3  $B \leftarrow$ TypeBaseline($S$, cfg) $S^\star \leftarrow S$
4  for $p \in \Pi$ do
5      $M \leftarrow p$.apply($S^\star, I$, cfg)  // context- and template-guided
6      if cfg.type_safe and $\neg$Accept($M, B$) then
7          continue  // reject and rollback to $S^\star$
8      $S^\star \leftarrow M$
9  if cfg.score_runtime then
10     RuntimeScore($S^\star$, cfg)
11 Emit($S^\star$, cfg) return $S^\star$
```

(iv) *Runtime evaluation and output:* After all passes have been applied, the final mutated code $S^\star$ can optionally be run in a sandbox or a different Python interpreter (as specified by cfg.runtime$_c$md, we choose MicroPython for our differential testing). The function RuntimeScore in Algorithm 5 executes the code with a time limit. If the program crashes (segmentation fault, interpreter panic) or times out, this is recorded (and a high score is assigned to that mutant, signaling a potentially interesting find). If it exits normally or with a benign error, a lower or zero score is assigned.

As shown in Algorithm 5, each transformation pass focuses on a certain *operator family* and uses the context index or templates to guide the mutation in a meaningful way. If a particular mutation is not valid under the type rules, the system does not terminate or fail; it simply skips that mutation and tries the next opportunity or next pass, which is important for robustness in an automated fuzzing setting.

The mutator provides four families of operators that act on CST while preserving syntactic validity (Table 6.2). **Structural/Block** operators rewrite statements or blocks to explore larger control and data-flow changes. **Peephole/Op** operators make local edits to operators, literals, and small expressions. **Semantic/Path** operators rewrite boolean logic and path conditions using common equivalences to steer execution. **Aggressive/Inflation** operators stress parser and runtime limits without breaking syntax. Some operators are only enabled when supported by the target (e.g., `WALRUS_INSERT`).

Structural operators include `REPLACE`/`ADD`/`DELETE`/`SWAP` for coarse edits, `REUSE/INJECT/COMBINE` for controlled code reuse, and `TRY_WRAP` for adding exception contexts; `INLINE_TEMP`/`EXTRACT_TEMP` perform simple refactorings. Peephole operators adjust arithmetic and boolean atoms (`ARITHMETIC_FLIP`, `LOGICAL_NEGATE`, `BOUNDARY_OFF_BY_ONE`), toggle decorators, and tweak default parameters. Semantic operators rewrite standard idioms (`SEMANTIC_AWARE`), flip conditions (`PATH_CONDITION`, `DEMORGAN`), and optionally insert the assignment expression (`WALRUS_INSERT`) when available. The `CHAOS` operator inflates literals, collections, and expression depth to probe resource limits while keeping the program valid.

Table 6.2: Mutation operators implemented in the mutator.

Family	**Operator**	**Effect**
Structural / Block	`REPLACE`	Replace node using template or context pool.
	`ADD`	Insert simple statements to blocks (diversity).
	`DELETE`	Remove node; keep `pass` if block empties.
	`SWAP`	Swap sibling statements (module or nested blocks).

Continued on next page

Table 6.2 – continued from previous page

Family	**Operator**	**Effect**
	`REUSE`	Replace with deep-clone from in-file pool.
	`INJECT`	Replace with deep-clone from secondary file.
	`COMBINE`	Combine templates/subtrees (conservative).
	`TRY_WRAP`	Wrap stmt/CF in `try/except Exception: pass`.
	`INLINE_-TEMP/EXTRACT_TEMP`	Basic temp var rewrites (peephole-ish).
Peephole / Op	`ARITHMETIC_FLIP`	Swap `+` $\leftrightarrow$ `-`, `*` $\leftrightarrow$ `//`, etc.
	`LOGICAL_NEGATE`	Insert/remove `not` around boolean exprs.
	`BOUNDARY_OFF_BY_-ONE`	± 1 tweaks of integer comparators.
	`DECORATOR_TOGGLE`	Cycle `staticmethod` $\rightarrow$ `classmethod` $\rightarrow$ `property`.
	`PARAM_DEFAULT_-MUTATE`	Nudge parameter default values (int/string).
Semantic / Path	`SEMANTIC_AWARE`	`len(x)==0` $\leftrightarrow$ `not x`; `is None` $\leftrightarrow$ `== None`; identity arith.
	`PATH_CONDITION`	In tests only: flip `and`/`or`, flip relations, ± 1 thresholds.
	`DEMORGAN`	Apply De Morgan's laws with safe parentheses.

Continued on next page

Table 6.2 – continued from previous page

Family	**Operator**	**Effect**
	`WALRUS_INSERT`	Insert `:=` in if/while/boolean sub-exprs (CPython).
Aggressive / Inflation	`CHAOS`	Huge ints/strings/bytes, grow collections, deepen expr, add blocks.

Overall, the context- and type-aware mutator provides a robust methodology for generating program variants. It balances **exploration** (through aggressive and chaotic mutations that can uncover edge cases) with **soundness** (through context awareness and optional type checking to keep mutants valid and interpretable).

6.4.2 Layer 2: Execution and Runtime Observation

The second layer executes each program from the corpus on both the baseline and CHERI-enabled builds of MicroPython and records detailed, normalized telemetry.

Each interpreter process is launched in a strict sandbox that enforces resource limits using `setrlimit` on CPU time, address space, and file writes. This prevents non-terminating programs from stalling the framework and contains side effects. The harness captures the process exit code, any terminating signal number, and the complete `stdout` and `stderr` streams. In addition, the CHERI harness logs detailed information about any capability faults (such as bounds, tag, or permission violations) that occur during execution.

6.4.3 Layer 3: Differential Analysis and Triage

The final layer analyses the paired execution records to find meaningful discrepancies, shrinks the inputs that cause them, and feeds the results back into the system.

6.4.3.1 Differential Oracle Logic

The comparator implements a state machine to classify each pair of outcomes. The primary classification of interest is a **Memory-Safety Differential**. This is triggered under two main conditions:

1. **Crash vs. Fault:** The baseline builds crashes with a generic memory signal (e.g., `SIGSEGV`, `SIGBUS`, while the CHERI build terminates cleanly with a specific capability fault signal. This indicates a memory error that CHERI precisely identifies.
2. **Success vs. Fault (Latent Bug):** The baseline build runs to completion (exit code 0) and produces some output, while the CHERI build terminates with a capability fault. This is a highly valuable finding, as it uncovers a latent memory safety violation that does not cause a crash in a conventional environment but is nonetheless a serious bug.

The other categories include: benign, semantic differential, timeout, and unknown. Benign cases are those where both builds exit cleanly with the same code and similar output. Semantic differentials occur when both builds complete but produce different outputs or error codes, indicating a logic bug rather than a memory safety issue. Timeouts are cases where one or both builds exceed the time limit, and unknown cases cover any other discrepancies not fitting the above categories.

6.5 Evaluation

Research Question 4: *As firmware complexity grows, traditional fuzzing struggles with highly structured inputs like language interpreters. How can we evolve test generation beyond simple mutation to rigorously assess architectural defences, thereby measuring the shift from vulnerability discovery to prevention?*

Concretely, we assess whether our test generation framework, which employs Large Language Models and Concrete Syntax Trees, combined with differential execution across CHERI and conventional MicroPython architectures, allows us to (1) uncover vulnerabilities related to memory safety within the interpreter and (2) demonstrate instances where CHERI mitigates crashes that manifest in the baseline build.

6.5.1 Experiment Setup

Our experimental design uses differential testing to evaluate the impact of the CHERI architecture on the MicroPython interpreter. To achieve this, we established two parallel execution environments. The first is a control environment, which runs a standard, non-CHERI build of MicroPython on a conventional Linux system. This setup provides a baseline for the interpreter's expected behaviour without hardware-based memory safety enhancements. The second is the experimental environment, where MicroPython is compiled for and executed on the CHERI-enabled Morello platform. This allows us to assess how CHERI's hardware-enforced memory safety influences the interpreter's behaviour under identical test conditions.

Our testing methodology accounts for several key variables. Since the CHERI build of MicroPython does not support the libffi module, we conduct separate test runs on the baseline system both with and without this module. This ensures that our comparisons accurately isolate the effects of the CHERI architecture. In addition, we run all test cases against the latest official version of MicroPython (1.27-preview) to help distinguish between pre-existing bugs in the core project and unique issues discovered during our analysis.

A custom harness is deployed in both environments to automate test execution and data collection. Each harness is responsible for running the same unified set of test scripts, enforcing resource limits like timeouts and memory usage, and capturing detailed telemetry. The data collected includes standard output and error streams, exit codes, and operating system signals. For the CHERI environment, the harness also records specific information about any capability violations, offering direct insight into hardware-level memory protection events.

6.5.2 Testcase Generation

The testcase generation process is a critical component of our evaluation framework, designed to produce a diverse and comprehensive set of test scripts that effectively probe the MicroPython interpreter for memory safety issues. This process begins with a curated corpus of seed scripts, which are derived from known vulnerabilities, bug reports, and common programming patterns that are likely to trigger memory-related errors.

The generation process employs a combination of techniques to expand this initial corpus. First, we use the LLM model to learn the structure and characteristics of the seed scripts, enabling it to generate new scripts that are syntactically valid and semantically relevant. The LLM is prefix-tuned to focus on constructs that are particularly pertinent to memory safety, such as pointer manipulations, buffer operations, and dynamic memory allocations.

Additionally, we leverage a context-aware mutator built on LibCST, as described in Section 6.4.1. This mutator applies a series of sophisticated mutations that respect the syntactic and semantic context of the code, ensuring that the generated scripts remain valid Python programs. The mutator can operate in both type-safe and non-type-safe modes, allowing us to explore a wide range of potential memory safety issues, including those that may arise from type inconsistencies.

The final corpus of test scripts is a blend of the original seed scripts and the newly generated variants. Each script is designed to be executed in both the baseline and CHERI-enabled environments, allowing for direct comparison of their behaviour. The diversity of the corpus is crucial, as it increases the likelihood of uncovering subtle memory safety violations that may not be apparent in more straightforward test cases.

We generated a total of 3800 testcases using the methods described above. These testcases were derived from an initial corpus seed, which were expanded through a combination of rule-based transformations and context-aware mutations. The resulting corpus encompasses a wide range of programming constructs and patterns, designed to thoroughly exercise the MicroPython interpreter's memory management capabilities.

We categorize the generated testcases into the same categories as the collection of CVE PoC and bug reports, shown in table 6.1. The categories include: raw memory, buffer protocol & view lifetime; binary conversions & biginit conners, ffi & native emitters, parsers, codecs & compressors in C, Filesystem, VFS& Race conditions, MMIO& peripherals,

Table 6.3: Distribution of testcases across different categories.

Category	**Generated testcases**
Raw memory, buffer protocol & view lifetime	702
Binary conversions & bigint corners	783
FFI / native emitters	401
Parsers, codecs & compressors in C	638
Filesystem, VFS & Race Conditions	682
MMIO & peripherals (embedded targets)	170
Interpreter internals, exceptions & GC	590
Total	**3800**

and interpreter internal, exceptions & GC. These caterories cover a broad spectrum of memory safety issues, ensuring that our evaluation is comprehensive and targets the most relevant areas of the interpreter's functionality. The distribution of testcases across these categories is shown in table 6.3.

6.5.3 Results

After generating the testcases, we executed the testcases on these three distinct targets: the baseline non-CHERI build of MicroPython v1.20, the CHERI-enabled build of MicroPython v1.20, and the latest development version of MicroPython (1.27-preview). This section details the findings, beginning with the raw crash counts, followed by our deduplication methodology, and concluding with an analysis of the unique bugs identified.

Table 6.4: The bug distribution

Category	**MicroPython v1.20 on non-CHERI**	**MicropPthon v1.20 on CHERI**	**MicroPython newest version**
Raw memory, buffer protocol & view lifetime	222	232	191
Binary conversions & bigint corners	115	165	99
FFI / native emitters	107	0	80
Parsers, codecs & compressors in C	15	11	4
Filesystem, VFS & Race Conditions	17	17	4
MMIO & peripherals	0	36	1
Interpreter internals, exceptions & GC	10	18	11
Total	**486**	**479**	**390**

6.5.3.1 Initial Crash Analysis

The initial execution of the test suite produced a large number of crashes across all targets. As summarized in table 6.4, we initially recorded 486 crashes on the non-CHERI build, 479 on the CHERI-enabled build, and 390 on the newest version of MicroPython. The ”FFI / native emitters” category highlights a key difference, as the CHERI build lacks libffi support and thus produced no crashes in this area. Conversely, the CHERI build detected a significant number of bugs in categories sensitive to memory layout, such as ”Binary conversions & bigint corners” (165 vs. 115) and ”MMIO & peripherals” (36 vs. 0).

6.5.3.2 Crash De-duplication Analysis

These initial crash counts are inflated by duplicate testcases. The generative and mutational nature of our fuzzer often produces many slight variations of a test case that all trigger the same underlying bug. To obtain an accurate count of unique vulnerabilities, we implemented an automated crash de-duplication pipeline.

The pipeline process each crash as follows:

1. Filter for Crashes: Only test runs resulting in a crash (defined by non-zero and non-one exit codes) are retained for analysis.
2. Extract Stack Trace: The GDB debugger is used to generate a backtrace for each crash.
3. Generate Crash Signature: We parse the backtrace to create a unique ”crash signature.” By default, this signature is defined by the top-most function call within the MicroPython source code (e.g., mp_obj_subscr @ obj.c:538). Frames from external libraries, such as libc, are ignored to ensure the signature is specific to the project’s code.

All test cases that produce the same crash signature are grouped together and counted as a single, unique bug.

6.5.3.3 Unique Bug Analysis

Applying this de-duplication process reveals a more precise landscape of bugs discovered, as shown in table 6.5. In total, we identified 47 unique bugs on the non-CHERI build (35 non-libffi, 12 libffi-related) and 43 unique bugs on the CHERI build. The newest version of MicroPython contained 35 unique bugs (24 non-libffi, 11 libffi-related).

When comparing the core interpreter (excluding libffi), the CHERI-enabled build exposed more unique memory safety bugs than the baseline non-CHERI build. This result is consistent with CHERI’s design, as its hardware enforced capability bounds and permissions transform latent memory errors, which may not cause a crash on a conventional architecture, into observable faults.

Table 6.5: The unique bug distribution after cleaning up the duplicates.

Target	bug-non libffi	bug-on libffi	Total
MicroPython v1.20 on non-CHERI	35	12	47
MicroPython v1.20 on CHERI	43	0	43
MicroPython v1.27 preview	24	11	35

Additionally, there are also differences in the number of testcases that trigger each unique bug, which also proves the CHERI's capability to catch more memory safety bugs. Figure 6.3 illustrates this by comparing trigger counts for the top 10 bugs identified on the CHERI build. For several bugs related to memory access, the detection gap is stark. For instance, a bug in *array_subscr()* was triggered by 64 testcases on CHERI but was never detected on the non-CHERI build. Similarly, a bug in *machine_mem_subscr()* was found by 32 testcases on CHERI and none on the baseline.

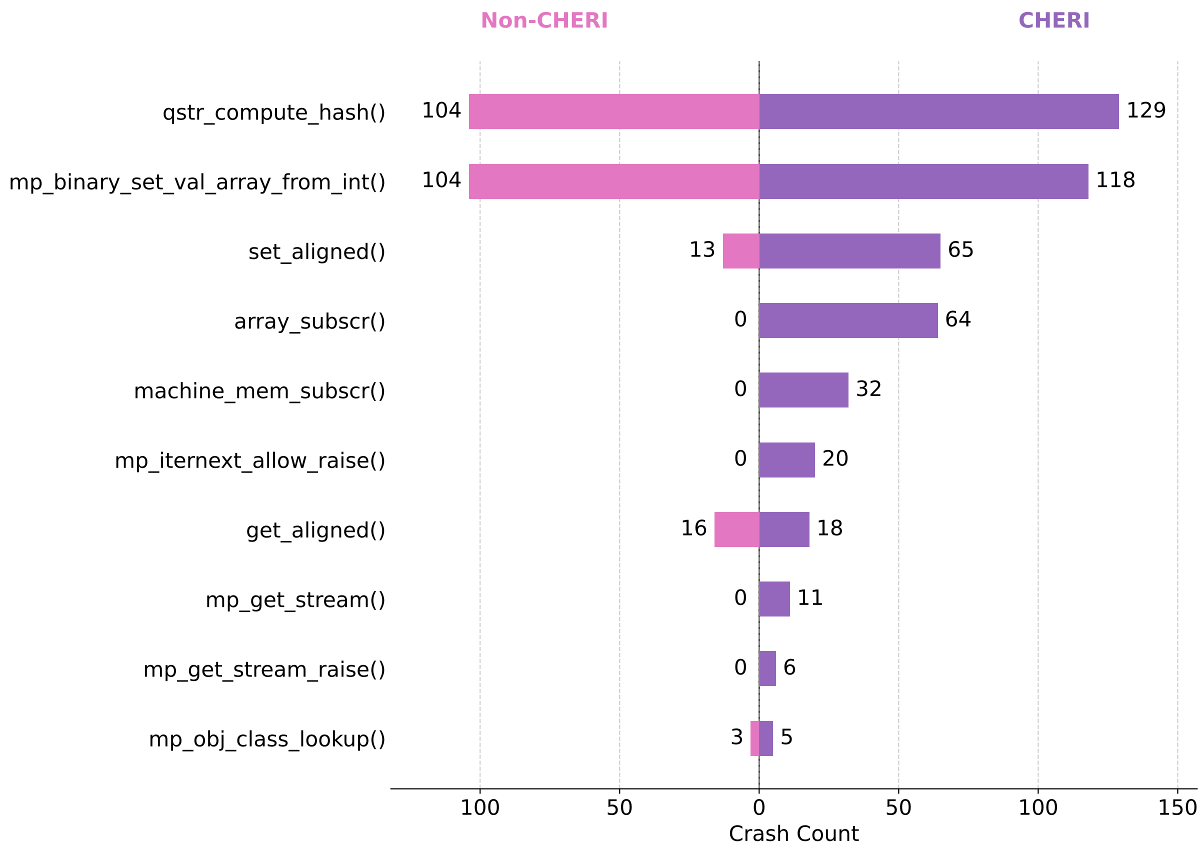


Figure 6.3: The distribution of bugs found on CHERI-build and non-CHERI build.

The differences between the CHERI-build and non-CHERI-build bugs are shown in table 6.5. It proves the CHERI-build can find more unique bugs. Even for the same bugs, there are more testcases can be detected on the CHERI-build and missed on the non-CHERI-build. It shows on figure 6.3. We select the top 10 bugs that we have found on CHERI-build MicroPython. The pink bars show the number of testcases that can trigger the bugs on non-CHERI build, and the purple boards show the number of testcases that can trigger the bugs on CHERI-build.

In these examples, CHERI-build produces almost twice as many triggering testcases overall. The largest gaps occur in paths that perform bounds, alignment, or pointer-derived access (e.g., array subscripts and aligned loads/stores). On CHERI, these operations fail fast at the exact misuse site (capability bounds or alignment checks), while the non-CHERI build only crashes if the corruption later propagates into illegal access.

Even when a bug is detectable on both platforms, the CHERI build consistently identifies it more frequently. For example, a bug in `set_aligned()` was triggered by 68 testcases on CHERI compared to only 13 on the non-CHERI build. This increased detection rate occurs because CHERI fails fast at the precise point of misuse, like a capability bounds or alignment check. In contrast, the non-CHERI build only crashes if and when the initial memory corruption later propagates to cause a segmentation fault or other fatal error.

6.5.4 Selected Bug Examples

MicroPython provides low-level primitives such as `uctypes`, the buffer protocol, and VFS block drivers that allow direct memory access in Python. In a conventional non-CHERI build, common bugs like out-of-bounds (OOB) accesses or the use of stale pointers often lead to silent memory corruption. The program may continue to execute, only to fail unpredictably at a later time, or in some cases, not fail at all, producing incorrect results. In contrast, CHERI enforces pointer bounds and permissions in hardware, providing robust spatial and temporal memory safety. When code attempts to use a pointer outside its designated range or after the memory it references has been deallocated (a stale pointer), CHERI immediately raises a precise fault at the point of the illegal access. This section presents concrete examples in MicroPython where CHERI reports bugs deterministically, while the non-CHERI Unix port typically does not.

6.5.4.1 Bug Example on CHERI-build MicroPython

The following examples demonstrate memory safety violations that are instantly caught by the CHERI architecture.

```python
import uctypes
owner = bytearray(b"hello")
base = uctypes.addressof(owner)
try:
  # Create an alias that is 8 bytes longer than the owner buffer
  raw = uctypes.bytearray_at(base, len(owner)+8)
  # Write into the memory beyond the owner's true boundary
  raw[len(owner):] = b"X"*8
  print("CRASH_SIG")
except Exception as e2:
  ok("passed", ty pe(e2).name)
```

Listing 6.1: Writing beyond the buffer via an overlong alias

In Listing 6.1, a raw memory alias is created that extends beyond the true boundary of the owner buffer. On the CHERI build, the first attempt to write past the end of the owner buffer raises a capability bounds fault, immediately terminating the illegal operation. On the non-CHERI build, this writes proceeds silently, corrupting adjacent heap memory and allowing the program to continue in an undefined state.

```python
import uctypes as u
owner = bytearray(16)
base  = u.addressof(owner)
#Create an alias starting 1 byte before the owner
a = u.bytearray_at(base - 1, 8)
a[0] = 1
print("DONE")
```

Listing 6.2: Writing before the buffer via an underflow alias

Listing 6.2 demonstrates a spatial memory violation where an alias is created starting just before the allocated buffer. The CHERI build faults on the write to a[0], as it is outside the valid bounds of the owner capability. Conversely, the non-CHERI build executes this write, corrupting memory preceding the buffer, and the program continues, ignores the error.

In Listing 6.3, a writable *memoryview* is created for a *bytearray*. The *bytearray* is then extended, a process that often requires reallocating its memory buffer to a new, larger location. This relocation invalidates the original *memoryview*.

```python
import os
path = "tmp.bin"
with open(path, "wb") as f:
  f.write(bytes(range(16)))
ba = bytearray(b"abcdefgh")   # 8 bytes
mv = memoryview(ba)           # Create a writable view
ba.extend(b"Z" * 4096)
with open(path, "rb") as f:
  n = f.readinto(mv)        # Attempt to write data via stale view
print("READ", n)
os.remove(path)
```

Listing 6.3: Stale memoryview used as a target for readinto

The CHERI build MicroPython detects that *mv* is a stale capability pointing to deallocated memory and faults when *f.readinto(mv)* is called. However, the non-CHERI build completes the *readinto* call successfully, writing data into a freed heap chunk that may have since been reallocated for another purpose, leading to silent heap corruption.

6.5.4.2 Bugs Examples on Latest MicroPython

This section shows four crash examples reproduced on the latest standard MicroPython Unix port. The first three reveal runtime defects in how MicroPython manages the lifetime of memoryview objects and handles reentrancy in list.sort(). The fourth example is a negative control, a Viper-compiled store to an unmapped memory address, which serves to validate our fault analysis methodology. All results in this section were obtained on a non-CHERI build.

```python
#Keep writable views to a small bytearray, then grow it so its
   storage moves.
#Writes through the old views corrupt unrelated runtime data.
import uctypes, gc
ba = bytearray(b"abcdefghij")          # 10 bytes
old = uctypes.addressof(ba)
views = [memoryview(ba) for _ in range(4)]
#Growth likely moves the backing buffer.
ba[:] = ba + b"X"*256
new = uctypes.addressof(ba)
print("MOVED?", old != new, hex(old), "->", hex(new))
gc.collect()
#Writes through stale views (still pointing at the old storage).
```

```
for mv in views:
  mv[0:1] = b"Y"
print("DONE")
```

Listing 6.4: Persistent writable views cause a crash after owner growth

The program typically prints MOVED? True, indicating the buffer was reallocated, and then crashes at a later, unrelated point in its execution. The backtrace often shows a fault during an indirect call within the virtual machine (e.g., in `mp_load_method_maybe`), which is consistent with the corruption of an object's type information or method table. MicroPython does not invalidate or prevent the use of existing writable *memoryview* objects when their underlying buffer is reallocated. When the *bytearray* grows, its storage moves, but the views retain a stale pointer to the old, now-freed memory location. Subsequent writes through these stale views corrupt whichever object has since been allocated in that memory space.

```

import uctypes, os

path = "t.bin"
with open(path, "wb") as f:
  f.write(bytes(range(16)))

a = array('I', [0]*4)       # 4 * 4B = 16 bytes
old = uctypes.addressof(a)
mv  = memoryview(a)         # Writable view
a.extend([1]*2048)          # Likely moves storage
new = uctypes.addressof(a)
print("MOVED?", old != new, hex(old), "->", hex(new))

with open(path, "rb") as f:
  n = f.readinto(mv)       # OS writes into stale address
print("READ", n)
```

```
os.remove(path)
```

Listing 6.5: Typed array view kept across growth; readinto writes via stale view

A crash typically occurs when the program exits the with block. The backtrace points to a fault within *mp_stream_close()* while trying to read the type field of what it assumes is a valid stream object, indicating that the object's metadata has been corrupted. A writable *memoryview* of a typed array persists after the array's storage is moved. This stale view is then used as the destination buffer for *readinto*, which writes data to the deallocated memory region, corrupting heap objects that now occupy that space.

```
import gc
class Evil:
def init(self, arr_ref):
  self.arr_ref = arr_ref
def lt(self, other):
  self.arr_ref.clear()  # Mutates the container being sorted
  gc.collect()          # Encourages immediate reuse of freed
   space
  return True
a = []
a.extend(Evil(a) for _ in range(5))
a.sort()
```

Listing 6.6: Comparator clears the list during sort operation

The virtual machine crashes with a segmentation fault inside mp_binary_op() while executing the quicksort algorithm. The GDB backtrace reveals that the program attempts to dereference a NULL pointer (rdi == 0) when trying to retrieve an object's type, indicating that the sort algorithm is operating on an invalid list element.The list.sort() implementation is not reentrant-safe. It does not protect against scenarios where the comparison function (__lt__) modifies the list being sorted. In this example, the comparator clears the list, deallocating its elements. The sort function, unaware of this change, proceeds to use its now-invalidated pointers to these elements, resulting in a NULL pointer dereference.

```python

@micropython.viper
def pokehi():
  p = ptr8(1 << 31)  # 0x80000000
  p[0] = 1           # Store to unmapped address
pokehi()
```

Listing 6.7: Viper writes to 0x80000000 (unmapped on Linux)

The program receives a SIGSEGV signal precisely at the instruction attempting to store a byte to the high memory address 0x80000000. This behaviour is expected on a Linux system where this address is part of an unmapped memory region.

6.6 Limitations and Future Work

Our differential testing framework has shown promising results in uncovering memory safety issues in the MicroPython interpreter. We also prove that CHERI's hardware-enforced memory safety can catch more latent bugs that do not cause crashes on conventional architectures. However, there are several limitations and areas for future improvement:

A: Testcase Diversity and Coverage. Although our testcase generation methods produce a wide range of inputs, there are still lots of similar testcases that trigger the same bugs. Ensuring diverse programs that exercise truly distinct code paths remains an open challenge [163]. Generating complex valid testcases for MicroPython is non-trivial. Other program generators like Csmith [164] and RustSmith [165] use careful rules to produce diverse, semantics-respecting programs. Adopting such rules from these generators could help improve the quality of the generated testcases. In the context of MicroPython, a more sophisticated generator could exercise rarely-used built-in types, error handlers, or interpreter internals that our current seeds and mutators might miss. Using code from real-world scripts or diverse libraries could further increase coverage of edge cases.

B: The more advanced generation technique. Our use of a prompt-guided large language model (LLM) proved effective for synthesizing valid Python programs, but there is room for more powerful generation approaches. Recent research suggests that LLMs can be harnessed as intelligent fuzzing agents-not only generating code, but also mutating and steering tests towards suspicious patterns. Future work could explore fine-tuning an LLM on MicroPython's grammar or known bug patterns to create an even more targeted generator. Another direction is to integrate semantic feedback into generation: for example, using the interpreter's own parser or a static analyser to guide the LLM to problematic constructs (e.g. deep recursion, large nested data structures, risky use of the C API). This could mitigate the risk of the model producing many syntactically different but semantically similar tests. Furthermore, an autonomous LLM-based agent might iteratively generate and refine tests by observing the differential outcomes-a form of AI-driven closed-loop fuzzing. Leveraging such generative AI techniques, which are starting to emerge in software testing research, could significantly enhance the breadth and depth of our test corpus.

C: Semantic Analysis and Dynamic Instrumentation. Our current differential oracle simply compares final outcomes (crash vs. capability fault vs. normal output). A deeper semantic analysis of execution could uncover subtle discrepancies and guide test generation. In future work, we envision integrating dynamic instrumentation tools like DynamoRIO [166] to trace memory accesses, pointer metadata, and control-flow coverage in detail during each test run. For example, logging every memory allocation and deallocation in the interpreter could help detect temporal safety issues even if they do not immediately lead to a crash.

D: Broader Target Support. Our initial study focused on MicroPython within a Unix-like environment and on the ARM Morello board. To assess the broader applicability of our approach, we intend to deploy our differential testing framework on other microcontroller boards. Moreover, our methodology could be extended to evaluate alternate embedded interpreters, such as CircuitPython, Cruby.

6.7 Conclusion

This chapter presented a novel differential testing framework that combines smart test generation with hardware-assisted checking to evaluate memory safety in MicroPython. We addressed RQ4 by showing how to evolve fuzz testing beyond blind mutation in order to rigorously assess an architectural defence (the CHERI capability system). Our approach leveraged an LLM-based generator and a LibCST mutator to produce semantically valid, high-risk Python programs, overcoming the limitations of naive fuzzers that often generate invalid code. We then ran each program on two builds-one standard and one CHERI-enhanced-under a unified harness that captures and normalizes their behaviour. By treating the CHERI-enabled interpreter as a variant implementation, we turned divergent outcomes into a powerful oracle for memory errors. Crucially, the CHERI build's precise traps transformed many elusive bugs into deterministic faults, providing immediate evidence of out-of-bounds accesses, use-after-free, and other unsafe actions that would otherwise go undetected. The experimental results demonstrate the effectiveness of this approach. The differential fuzzer discovered huge gaps in detecting memory safety bugs between CHERI and non-CHERI builds. Additionally, we also uncovered 35 unique bugs in the latest version of MicroPython. All the code and discovered bugs have been responsibly disclosed to the MicroPython team, shown in Appendix B. We also provide the source code for differential framework and MicroPython bug database to the community.[3]

3. https://github.com/MaksimFeng/ML4Secure/tree/evolve

Chapter 7

Conclusions

7.1 Summary of Contributions

This thesis advances automated vulnerability discovery in IoT firmware by designing and evaluating four complementary fuzzing techniques, each addressing a core challenge highlighted in chapter 1. Taken together, the results show that domain-aware strategies, richer feedback signals, hardware-assisted execution, and structured test generation can materially improve the effectiveness of fuzzing for embedded systems. The evaluations span industrial control systems, general-purpose firmware, and a memory-safe architecture, and show that a hardware-centric approach can both increase the rate at which bugs are found and support the assessment of preventative security mechanisms.

Sizzler introduces a domain-specific, learning-based fuzzer for ladder logic in programmable logic controllers (PLCs). It uses a SeqGAN model to learn mutation policies from the fuzzing process so that generated inputs bypass PLC-specific checks and reach deeper logic. Sizzler emulates ladder diagrams across diverse microcontroller platforms using a refined QEMU backend. In practice, Sizzler uncovered a critical buffer overflow in the OpenPLC runtime (CVE-2023-43184) that allowed crafted ladder code to inject payloads via PLC slave attributes and crash the controller. Across our dataset, Sizzler found multiple PLC logic issues and achieved higher coverage on standard embedded benchmarks (Magma and LAVA-M) than baseline fuzzers. These findings answer **RQ1**: domain-specific learning improves mutation effectiveness, enabling inputs that pass PLC-specific filters and expose deeper code paths that general-purpose fuzzers miss. Sizzler's success on closed-source PLC binaries underscores the value of combining domain knowledge with learning-based mutation for IoT firmware.

FuzzRDUCC contributes a data-flow guided feedback mechanism for binary fuzzing, answering **RQ2** by directly comparing def-use coverage with classic control-flow (edge) coverage in firmware and driver contexts. Integrated into QEMU at the translation level, FuzzRDUCC instruments binaries to record where values are defined and where they are later used, not only which basic blocks execute. Our evaluation shows that def-use coverage provides stronger guidance to solve hard conditional checks and magicbyte comparisons. Over 24-hour campaigns on real binaries (e.g., GNU Binutils utilities), FuzzRDUCC achieved faster coverage growth than AFL++ and several state-of-the-art binary-only fuzzers, including UAFuzz and ZAFL, thereby exploring paths that edge coverage alone did not reach. Notably, it revealed a unique null-pointer dereference in the *strip* utility's relocation-handling logic (within *copy_relocations_in_section*) by manipulating specific relocation entries, whereas AFL++'s generic mutations failed to pass initial format checks and missed this code region entirely. While total crash counts did not always scale with coverage—AFL++ still found certain bugs faster—the quality of coverage from FuzzRDUCC exposed new bug classes (e.g., pointer misuse in *strip*) that edge-based fuzzers overlooked. These results support the claim that enriching feedback with data-flow information can guide test generation more effectively than standard edge coverage, especially for firmware with complex, input-dependent logic.

Hardfuzz answers **RQ3** by showing that on-device fuzzing with hardware breakpoints can deliver high-fidelity execution and effective feedback at practical speed on microcontrollers. The framework runs target firmware on real hardware and uses the built-in debug interface to gather feedback, avoiding the uncertainties of emulation. Hardfuzz guides fuzzing with def-use insights (as in FuzzRDUCC) while operating under the constraint of a small number of hardware breakpoints. Inspired by recent "debugger-driven" fuzzing techniques that use GDB and on-chip debug hardware [40], we show that even 4–6 breakpoints on typical Cortex-M MCUs are sufficient for coverage-guided, on-device fuzzing. By placing breakpoints along high-value def-use chains and relocating them iteratively, Hardfuzz explores new code while executing each test at native MCU speed (avoiding the $>10\times$ slowdowns common in full emulation). The approach is fast, accurate, and non-intrusive: all peripheral interactions and timing behaviour occur as on the actual device, and any crash is a true device crash. In our evaluation, Hardfuzz achieved higher coverage and more unique basic blocks than GDBFuzz under QEMU emulation, indicating that our breakpoint assignment strategy is efficient. For the real MCUs, Hardfuzz outperformed GDBFuzz in coverage finding, demonstrating that on-device fuzzing with data-flow guidance is both practical and effective. These outcomes confirm **RQ3**: fuzzing on real hardware via debug breakpoints is practical and can match or exceed emulation-based approaches in both coverage and bug finding.

Finally, **MicroPython differential fuzzing** addresses **RQ4** by moving beyond pure bug hunting to the structured evaluation of defensive mechanisms. We present a differential fuzzing framework for MicroPython that generates valid, structured test cases using LLMs and CST mutations. We then run the same tests in two environments: (i) a CHERI-capability-enabled MicroPython interpreter on a CHERI system and (ii) a baseline interpreter on a conventional CPU. By comparing outcomes (e.g., crashes, exceptions, silent execution), we measure how many memory errors are prevented by CHERI. Many inputs produced divergent behaviour: heap overflows and use-after-free errors that caused silent memory corruption on the baseline were safely trapped on CHERI as bounds violations. In several cases, the CHERI-enabled interpreter halted latent memory safety bugs that had not been observable on conventional hardware until fuzzing uncovered them. CHERI detected 8 unique memory safety violations which can not be detected on the baseline system.Thus, **RQ4** is answered: when systems include strong architectural safety features, fuzzing can and should be used to assess these preventative measures. With LLM-assisted, high-structure test generation, we achieved broad interpreter coverage and showed how architectural memory safety (CHERI capabilities) can eliminate entire classes of vulnerabilities that remain exploitable on traditional architectures.

7.2 Limitations and Future Work

While these contributions advance automated IoT firmware testing, several limitations affect generality and scalability of our techniques.

Vendor specificity in PLCs. Sizzler currently targets ladder logic from open-source PLC environments, yet most industrial PLC binaries use proprietary formats. There is no universal emulator or intermediate representation for all PLC equipment, vendors differ in memory maps, instruction sets, and binary layouts. Although Sizzler shows the value of domain-specific learning for PLCs, broad adoption will require adaptation to diverse vendor ecosystems. Future work includes developing cross-vendor intermediate representations, applying automated reverse engineering to infer semantics, and building lifting pipelines that reduce per-vendor engineering effort.

Performance and overhead. Instrumentation-heavy techniques incur runtime costs. FuzzRDUCC's data-flow instrumentation introduces overhead in QEMU. We mitigate this with heuristics that select high-value def-use chains, but the prototype still runs slower than native execution. Sizzler's use of AFL++ on QEMU also adds overhead, full tracing in QEMU can introduce slowdowns on the order of 13×. Hardfuzz executes tests at device speed, but serial I/O and frequent breakpoint handling add latency, and the small number of hardware breakpoints limits simultaneous instrumentation. The stop–restart cycle for on-device testing lowers throughput relative to in-memory fuzzing. In short, richer feedback improves guidance but reduces executions per second, which can hinder scalability for very large firmware that require millions of test runs. Future optimizations include selective and adaptive instrumentation, use of hardware tracing features, parallelization across device farms, and hybrid fuzzing that switches between emulation and hardware based on feedback value.

LLM-generated structured inputs. Our CHERI+MicroPython pipeline depends on the quality and diversity of the underlying language model and prompts. While coverage was strong, we cannot claim complete exercise of all interpreter paths, especially rare edge cases. Moreover, our study focused on a single interpreter and one architectural mechanism (CHERI). Future work should extend to other interpreters and runtimes, and compare multiple safety mechanisms (e.g., memory tagging or sanitisation-based defences). Improvements to input generation—such as constraint-guided CST edits, semantic or type-aware mutations, and feedback that rewards specification conformance—could further increase depth and breadth of exercised behaviours.

In summary, the four systems deliver clear gains but involve trade-offs: limited generality for PLC fuzzing, runtime overhead for instrumentation, hardware resource constraints for on-device fuzzing, and reliance on external models for structured inputs. Addressing these constraints points toward a unified, scalable framework that (i) abstracts vendor-specific details, (ii) adapts instrumentation to maximise feedback per unit time, (iii) leverages device farms and hardware tracing to close the throughput gap, and (iv) uses semantics-aware generators that better align with target specifications.

7.3 Final Remarks

This thesis shows that fuzzing for IoT and embedded systems is strengthened by domain-specific knowledge, richer feedback, hardware-in-the-loop integration, and structured input generation. We frame the contributions along two axes: where the target runs (real MCUs, rehosting, emulation) and what guides exploration (data-flow and semantic feedback paired with grammar and model guided generators). Our results (new PLC vulnerabilities, previously unseen firmware bugs, and empirical evidence that architectural safety features such as CHERI can block whole classes of memory-safety errors) demonstrate significant security impact. The remaining gaps point to next steps: make the techniques more general across platforms, faster at scale, and easier to deploy. The broader goal is an automated, end-to-end fuzzing ecosystem that not only finds vulnerabilities across the diverse IoT landscape but also informs the design and evaluation of safer firmware and hardware, enabling a paradigm shift in the field from discovery to prevention of vulnerabilities.

Appendices

A Prompt for LLM Testcase Generation

This section provides the exact prompt we used for generating MicroPython testcases. The prompt is designed to guide the LLM to produce concise, syntactically valid, and semantically meaningful Python code snippets that can be executed in a MicroPython environment.

You are an expert MicroPython vulnerability hunter. Generate standalone .py tests for the MicroPython Unix port. Each test must be a single file that runs fast, is self-validating, and prints a result. Focus on finding memory-safety bugs (bounds violations, UAF, stale views, unchecked pointer use, corruptions) and hard VM failures (segfault/abort), not just `ValueError`/`MemoryError`.

Environment & constraints (must follow)

- Target: MicroPython Unix build.
- `memoryview` step must be 1 (use contiguous slices only).
- `bytearray.clear()` is absent → use `del ba[:]`.

- `bytearray *= N` is absent → grow with `extend` or slice-assign `ba[:] = ba + big_bytes`.
- Prefer `open()` over `os.open`; `fsync`/`truncate` may not exist.
- If a feature/module is missing, print `SKIP: <reason>` and exit cleanly.

Categories & tactics (generate N tests per category; default N=40)

A) Raw memory, buffer protocol & view lifetime

- **Aggressively exercise:**
  - `uctypes.bytes_at(addr,len)` and `bytearray_at(addr,len)` with negative/NULL/huge addresses.
  - `uctypes.struct(base, desc)` with misaligned fields, fields crossing buffer end, and negative base offsets.
  - `addressof()` on immutable `str`/`bytes`, then write via `bytearray_at` and observe consequences (e.g., content/CRC drift).
  - `Stale view` patterns: hold `memoryview`/`uctypes` alias; perform `bytearray.extend(huge)`, `ba[:] = ba + big`, `del ba[:]` (clear), or `array('B').extend(...)`; then write via the old view/alias.
  - Overlapped aliasing: multiple `bytearray_at` to overlapping ranges; write interleaved after owner growth.
- **Oracles:**
  - A crash (rc $\neq$ 0) or "strange success" like writing through a view after owner moved.
  - Post-write invariants: if you mutate `bytes`/`str` via raw writes, check `ubinascii.crc32`/prefix/equality changed unexpectedly.

B) Binary conversions & bigint corners

- **Push `ustruct`/`struct`:**

  - `pack_into`/`unpack_from` with **giant/negative offsets**, **misaligned** offsets into `memoryview(array('H'/'I'))`.
  - Huge repeat counts (`"999999x"`, `"1000000s"`) and composite formats with many fields (check integer overflow in size maths).
- `int.to_bytes/from_bytes`:
  - Extreme lengths (0, 1, huge), signed/unsigned mismatch; pipe into `*_into` to **write past end** of a small view.
- **Oracles:** crash; silent buffer corruption (verify guard bytes); diverging behaviour vs expected bounds checks.

D) Parsers, codecs & UTF-8

- `ujson`: deeply nested, massive numbers, invalid escapes, recursive structures to trigger edge walkers.
- `ubinascii`: `hexlify/unhexlify` with injected invalids mid-stream; large CRC32 on unaligned memoryviews.
- `ure`: catastrophic backtracking patterns that also manipulate big buffers nearby (look for mis-sized `memcpy`).
- `utf-8`: overlong/invalid continuations in bulk.
- **Oracles:** crash or inconsistent results after heavy decode loops.

E) Filesystem & VFS edges

- `readinto` into **typed** memoryviews (`array('H'/'I')`) with lengths not multiples of element size; chain partial reads into overlapping slices of the *same* buffer.
- Use **the same memoryview** interleaved with multiple file objects; rename/remove file between reads.
- After `readinto`, **drop or clear** the owner (`del ba[:]`, reassign `buf = bytearray(1)`), then attempt **view write**.
- **Oracles:** crash, stale view write success, data torn in overlapped regions.

F) MMIO (optional, skip if machine missing)

- Misaligned `mem8/16/32` accesses and addresses that cross 4 KiB windows.
- **Oracles:** bound/permission faults (on CHERI builds) or crashes on permissive ports.

G) Interpreter internals & GC interactions

- Finalizers that resurrect objects and allocate during GC; chained generators whose `close()` allocates in `finally`.
- Sorting with comparators that mutate/lengthen the list; schedule many callbacks that raise, interleaving `gc.collect()`.
- `micropython.heap_lock()` misuse: attempt allocations while locked (expect clean error, not crash).
- **Oracles:** crash or corrupted interpreter state.

FFI / C extensions / native emitters Targets

- Unix port: `ffi/ffilib` (`dlopen`/`dlsym` + calls)
- `@micropython.viper` / `@micropython.native` (where available)
- Inline asm on MCU ports (e.g., `asm_thumb`)

Test structure & output (strict)

- Each file must:
  - Import only what it needs.
  - Wrap the core in `try/except` and `log("OK"/"EXC", ...)`.
  - Avoid unsupported APIs per constraints above.
  - Prefer short, deterministic sequences over loops; add `thrash()` judiciously to shake lifetime bugs.
  - Print exactly one final line that starts with one of:
    - `OK`, `EXC <Type>`, `SKIP <reason>`, or a brief custom tag like `CRASH_SIG` if you detect inconsistency.
- Aim for **crash or deterministic invalid write**.

Seed patterns to instantiate across tests (cover systematically)

- **Addresses:** $[-2^{63}$ .. -1], [0], [1, 2, 16, 4096, 2^{31}] (clamped to platform).
- **Sizes:** [0, 1, 3, 7, 15, 31, 63, 127, 255, 4096, 1000000].
- **Offsets into pack/unpack:** $[-10^9$, -1, 0, 1, 2, 3, 4, 7, 15, 31].
- **Element widths:** 1/2/4; owners: `bytearray`, `array('B')`, `array('H')`, `array('I')`.
- **Owner growth ops:** `extend(big)`, `ba[:] = ba + big`, `del ba[:]`.
- **GC cadence:** call `thrash()` before/after a potentially stale deref.

Deliverable

- Produce N=40 tests per category (A,B,D,E,G; F only if machine exists).
- Each file name starts with `test_<categorycode><case#>_<brief>.py`.

Example code you can follow

```python
import gc
class Evil:
    def __init__(self, arr_ref):
        self.arr_ref = arr_ref

    def __lt__(self, other):
        self.arr_ref.clear()
        gc.collect()
        return True
a = []
evil_objs = [Evil(a) for _ in range(5)]
a.extend(evil_objs)
print(len(a))
a.sort()
print(len(a))
print(len(evil_objs))
```

```python
#---------------------------
import gc
b = bytearray(b"ABCD")
mv = memoryview(b)
try:
    b[:] = b""
except Exception as e:
    print("shrink blocked:", type(e).__name__, e)

for _ in range(2000):
    _ = bytearray(64)
gc.collect()
try:
    print("mv_head=", bytes(mv[:1]))
except Exception as e:
    print("mv read exc:", type(e).__name__, e)
```

B Bug Reports Submitted to the MicroPython Project

The following GitHub issues were opened as part of outcome of differential testing framework for MicroPython.

- Issue #18172
- Issue #18171
- Issue #18170
- Issue #18169
- Issue #18168

- Issue #18167
- Issue #18166
- Issue #17941

Bibliography

[1] Binbin Zhao et al. ‘A large-scale empirical analysis of the vulnerabilities introduced by third-party components in IoT firmware’. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 442–454.

[2] Zhichuang Sun et al. ‘OAT: Attesting operation integrity of embedded devices’. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1433–1449.

[3] Taylor Hardin et al. ‘Application memory isolation on ultra-Low-powerMCUs’. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 127–132.

[4] Alejandro Mera et al. ‘D-box: DMA-enabled compartmentalization for embedded applications’. In: *arXiv preprint arXiv:2201.05199* (2022).

[5] LLVM Project. *Clang Static Analyzer*. Accessed: 26 August 2025. 2025. url: `https://clang-analyzer.llvm.org/`.

[6] 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)*. 2020, pp. 1237–1254.

[7] Marius Muench et al. ‘Avatar 2: A multi-target orchestration platform’. In: *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* Vol. 18. 2018, pp. 1–11.

[8] Barton P Miller, Lars Fredriksen and Bryan So. ‘An empirical study of the reliability of UNIX utilities’. In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[9] Oliver Chang et al. 'OSS-Fuzz: Continuous fuzzing for open source software'. In: *URL: https://github. com/google/ossfuzz* (2016).

[10] Jiongyi Chen et al. 'IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing.' In: *NDSS*. 2018, pp. 1–15.

[11] Yaowen Zheng et al. 'FIRM-AFL:High-Throughput greybox fuzzing of IoT firmware via augmented process emulation'. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1099–1114.

[12] Meng Xu et al. 'Krace: Data race fuzzing for kernel file systems'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1643–1660.

[13] Zu-Ming Jiang, Jia-Ju Bai and Zhendong Su. 'DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation'. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 4949–4965.

[14] George Klees et al. 'Evaluating fuzz testing'. In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, pp. 2123–2138.

[15] Alejandro Mera et al. 'DICE: Automatic emulation of DMA input channels for dynamic firmware analysis'. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1938–1954.

[16] Christopher Wright et al. 'Challenges in firmware re-hosting, emulation, and analysis'. In: *ACM Computing Surveys (CSUR)* 54.1 (2021), pp. 1–36.

[17] Nassim Corteggiani, Giovanni Camurati and Aurélien Francillon. 'Inception:System-Wide security testing of Real-World embedded systems software'. In: *27th USENIX security symposium (USENIX security 18)*. 2018, pp. 309–326.

[18] Karl Koscher, Tadayoshi Kohno and David Molnar. 'SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems'. In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. 2015.

[19] Jonas Zaddach et al. 'AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.' In: *NDSS*. Vol. 14. 2014. 2014, pp. 1–16.

[20] Stefan Nagy and Matthew Hicks. 'Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing'. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 787–802.

[21] Michal Zalewski. *American Fuzzy Lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl 2017.

[22] Chenyang Lyu et al. 'MOPT: Optimized mutation scheduling for fuzzers'. In: *28th USENIX security symposium (USENIX security 19)*. 2019, pp. 1949–1966.

[23] Mingyuan Wu et al. 'One fuzzing strategy to rule them all'. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1634–1645.

[24] Yan Wang et al. 'A systematic review of fuzzing based on machine learning techniques'. In: *PloS one* 15.8 (2020), e0237749.

[25] Thiago Rodrigues Alves et al. 'OpenPLC: An open source alternative to automation'. In: *IEEE Global Humanitarian Technology Conference (GHTC 2014)*. IEEE. 2014, pp. 585–589.

[26] Peng Chen and Hao Chen. 'Angora: Efficient fuzzing by principled search'. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 711–725.

[27] Ahmad Hazimeh, Adrian Herrera and Mathias Payer. 'Magma: A ground-truth fuzzing benchmark'. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.

[28] Jonathan H Turner. 'Self, emotions, and extreme violence: Extending symbolic interactionist theorizing'. In: *Symbolic Interaction* 30.4 (2007), pp. 501–530.

[29] Ayushi Sharma et al. 'Rust for embedded systems: current state and open problems'. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2024, pp. 2296–2310.

[30] Van-Thuan Pham, Marcel Böhme and Abhik Roychoudhury. 'Aflnet: A greybox fuzzer for network protocols'. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 460–465.

[31] Jincheng Wang, Le Yu and Xiapu Luo. 'Llmif: Augmented large language model for fuzzing iot devices'. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, pp. 881–896.

[32] Paul Fiterau-Brostean et al. 'Analysis of DTLS implementations using protocol state fuzzing'. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2523–2540.

[33] Vaggelis Atlidakis, Patrice Godefroid and Marina Polishchuk. 'Restler: Stateful rest api fuzzing'. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 748–758.

[34] Xiaotao Feng et al. 'Detecting vulnerability on IoT device firmware: A survey'. In: *IEEE/CAA Journal of Automatica Sinica* 10.1 (2022), pp. 25–41.

[35] Alexander Bulekov et al. 'Morphuzz: Bending (input) space to fuzz virtual devices'. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1221–1238.

[36] Bo Yu et al. 'Poster: Fuzzing iot firmware via multi-stage message generation'. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 2525–2527.

[37] Zhenhao Luo et al. 'VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search.' In: *NDSS*. 2023.

[38] Roberto Natella. 'StateAFL: Greybox Fuzzing for Stateful Network Servers'. In: *Empirical Software Engineering* 27.191 (2022).

[39] Pedram Amini and Aaron Portnoy. *Sulley Fuzzing Framework*. https://github.com/OpenRCE/sulley. 2010.

[40] Max Eisele et al. 'Fuzzing embedded systems using debug interfaces'. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 1031–1042.

[41] Kevin Laeufer et al. 'RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs'. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[42] Yanhao Wang et al. 'Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.' In: *NDSS*. 2020.

[43] Andrea Fioraldi et al. 'AFL++ combining incremental steps of fuzzing research'. In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 2020, pp. 10–10.

[44] Joobeom Yun et al. ‘Fuzzing of embedded systems: A survey’. In: *ACM Computing Surveys* 55.7 (2022), pp. 1–33.

[45] Gen Zhang et al. ‘Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing’. In: *arXiv preprint arXiv:2401.15956* (2024).

[46] Yaowen Zheng et al. ‘Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation’. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* 2022, pp. 417–428.

[47] Xiaotao Feng et al. ‘Snipuzz: Black-box fuzzing of iot firmware via message snippet inference’. In: *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security.* 2021, pp. 337–350.

[48] Dokyung Song et al. ‘Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints’. In: *29th USENIX Security Symposium (USENIX Security 20).* 2020, pp. 2541–2557.

[49] Moritz Schloegel et al. ‘Sok: Prudent evaluation practices for fuzzing’. In: *2024 IEEE Symposium on Security and Privacy (SP).* IEEE. 2024, pp. 1974–1993.

[50] Jun Li, Bodong Zhao and Chao Zhang. ‘Fuzzing: a survey’. In: *Cybersecurity* 1.1 (2018), p. 6.

[51] Maialen Eceiza, Jose Luis Flores and Mikel Iturbe. ‘Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems’. In: *IEEE Internet of Things Journal* 8.13 (2021), pp. 10390–10411.

[52] Lukas Seidel, Dominik Christian Maier and Marius Muench. ‘Forming Faster Firmware Fuzzers.’ In: *USENIX Security Symposium.* 2023, pp. 2903–2920.

[53] Sebastian Österlund et al. ‘ParmeSan: Sanitizer-guided greybox fuzzing’. In: *29th USENIX Security Symposium (USENIX Security 20).* 2020, pp. 2289–2306.

[54] Yuseok Jeon et al. ‘FuZZan: Efficient sanitizer metadata design for fuzzing’. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20).* 2020, pp. 249–263.

[55] Joschua Schilling et al. ‘A binary-level thread sanitizer or why sanitizing on the binary level is hard’. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 1903–1920.

[56] Dae R Jeong et al. ‘Razzer: Finding kernel race bugs through fuzzing’. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 754–768.

[57] Dokyung Song et al. ‘Periscope: An effective probing and fuzzing framework for the hardware-os boundary’. In: *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society. 2019, pp. 1–15.

[58] Qiang Liu et al. ‘Videzzo: Dependency-aware virtual device fuzzing’. In: *2023 IEEE Symposium on security and privacy (SP)*. IEEE. 2023, pp. 3228–3245.

[59] Elia Geretto et al. ‘Snappy: Efficient fuzzing with adaptive and mutable snapshots’. In: *Proceedings of the 38th annual computer security applications conference*. 2022, pp. 375–387.

[60] Max Eisele et al. ‘Embedded fuzzing: a review of challenges, tools, and solutions’. In: *Cybersecurity* 5.1 (2022), p. 18.

[61] Alejandro Mera et al. ‘SHiFT: Semi-hosted Fuzz Testing for Embedded Applications’. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 5323–5340.

[62] Ren Ding et al. ‘Hardware support to improve fuzzing performance and precision’. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2214–2228.

[63] Romain Malmain, Andrea Fioraldi and Aurélien Francillon. ‘LibAFL QEMU: A library for fuzzing-oriented emulation’. In: *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*. 2024.

[64] Zhongwen Feng and Junyan Ma. ‘TWFuzz: Fuzzing embedded systems with three wires’. In: *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 2024, pp. 107–118.

[65] Wenqiang Li et al. ‘μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware’. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1–12.

[66] Tai Yue et al. 'Armor: Protecting software against hardware tracing techniques'. In: *IEEE Transactions on Information Forensics and Security* 19 (2024), pp. 4247–4262.

[67] James Patrick-Evans, Lorenzo Cavallaro and Johannes Kinder. 'POTUS: Probing Off-The-ShelfUSB Drivers with Symbolic Fault Injection'. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[68] Philip Sperl and Konstantin Böttinger. 'Side-channel aware fuzzing'. In: *European Symposium on Research in Computer Security*. Springer. 2019, pp. 259–278.

[69] Pallavi Borkar et al. 'WhisperFuzz:White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors'. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 5377–5394.

[70] Hangtian Liu et al. 'Labrador: Response guided directed fuzzing for black-box iot devices'. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, pp. 1920–1938.

[71] Yu Zhang et al. 'SRFuzzer: An automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities'. In: *Proceedings of the 35th annual computer security applications conference*. 2019, pp. 544–556.

[72] Prashast Srivastava et al. 'Firmfuzz: Automated iot firmware introspection and analysis'. In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. 2019, pp. 15–21.

[73] Roberto Natella. 'Stateafl: Greybox fuzzing for stateful network servers'. In: *Empirical Software Engineering* 27.7 (2022), p. 191.

[74] Hui Peng and Mathias Payer. 'USBFuzz: A framework for fuzzing USB drivers by device emulation'. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2559–2575.

[75] Kaiming Fang and Guanhua Yan. 'Emulation-instrumented fuzz testing of 4g/lte android mobile devices guided by reinforcement learning'. In: *European Symposium on Research in Computer Security*. Springer. 2018, pp. 20–40.

[76] Marius Muench et al. 'What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.' In: *NDSS*. 2018.

[77] Ferenc Speiser, István Szalay and Dénes Fodor. 'Embedded System Simulation Using Renode'. In: *Engineering Proceedings* 79.1 (2024), p. 52.

[78] Peter S Magnusson et al. 'Simics: A full system simulation platform'. In: *Computer* 35.2 (2002), pp. 50–58.

[79] Dominik Maier, Benedikt Radtke and Bastian Harren. 'Unicorefuzz: On the viability of emulation for kernelspace fuzzing'. In: *13th USENIX workshop on offensive technologies (WOOT 19)*. 2019.

[80] Mingeun Kim et al. 'Firmae: Towards large-scale emulation of iot firmware for dynamic analysis'. In: *Proceedings of the 36th Annual Computer Security Applications Conference*. 2020, pp. 733–745.

[81] Abraham A Clements et al. 'HALucinator: Firmware re-hosting through abstraction layer emulation'. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1201–1218.

[82] Tobias Scharnowski et al. 'Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing'. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1239–1256.

[83] Dennis Kengo Oka, Toshiyuki Fujikura and Ryo Kurachi. 'Shift left: Fuzzing earlier in the automotive software development lifecycle using hil systems'. In: *Proc. 16th ESCAR Europe* (2018), pp. 1–13.

[84] Jounghyuk Suh et al. 'Programmable analog device array (PANDA): A methodology for transistor-level analog emulation'. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 60.6 (2013), pp. 1369–1380.

[85] Hangwei Zhang et al. 'SIoTFuzzer: fuzzing web interface in IoT firmware via stateful message generation'. In: *Applied Sciences* 11.7 (2021), p. 3120.

[86] Patrick Cousineau and Brian Lachine. 'Enhancing boofuzz process monitoring for closed-source SCADA system fuzzing'. In: *2023 IEEE International Systems Conference (SysCon)*. IEEE. 2023, pp. 1–8.

[87] Kai Su et al. 'Fuzz Wars: The Voltage Awakens–Voltage-Guided Blackbox Fuzzing on FPGAs'. In: *2024 IEEE 42nd VLSI Test Symposium (VTS)*. IEEE. 2024, pp. 1–7.

[88] Seyed Mohammadjavad Seyed Talebi et al. 'Charm: Facilitating dynamic analysis of device drivers of mobile systems'. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 291–307.

[89] Dimitrios Tychalas, Hadjer Benkraouda and Michail Maniatakos. 'ICSFuzz: Manipulating I/Os and repurposing binary code to enable instrumented fuzzing in ICS control applications'. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2847–2862.

[90] Evan Johnson et al. 'Jetset: Targeted Firmware Rehosting for Embedded Systems.' In: *USENIX Security Symposium*. 2021, pp. 321–338.

[91] Wei Zhou et al. 'Automatic firmware emulation through invalidity-guided knowledge inference'. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2007–2024.

[92] Patrice Godefroid, Bo-Yuan Huang and Marina Polishchuk. 'Intelligent REST API data fuzzing'. In: *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 725–736.

[93] Zhijie Gui et al. 'Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution'. In: *Ieee Access* 8 (2020), pp. 29826–29841.

[94] Andrew Fasano et al. 'Sok: Enabling security analyses of embedded systems via rehosting'. In: *Proceedings of the 2021 ACM Asia conference on computer and communications security*. 2021, pp. 687–701.

[95] Wei Zhou et al. 'Automatic Firmware Emulation through Invalidity-guided Knowledge Inference'. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[96] Drew Davidson et al. 'FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution'. In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 463–478.

[97] Cristian Cadar, Daniel Dunbar, Dawson R Engler et al. 'Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.' In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[98] Yao Yao et al. 'Identifying privilege separation vulnerabilities in IoT firmware with symbolic execution'. In: *European Symposium on Research in Computer Security*. Springer. 2019, pp. 638–657.

[99] Changming Liu et al. 'CO3: Concolic Co-execution for Firmware'. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 5591–5608.

[100] Insu Yun et al. 'QSYM: A practical concolic execution engine tailored for hybrid fuzzing'. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 745–761.

[101] Nick Stephens et al. 'Driller: Augmenting fuzzing through selective symbolic execution.' In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[102] Hongliang Liang et al. 'A Practical Concolic Execution Technique for Large Scale Software Systems'. In: *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. 2020, pp. 312–317.

[103] Jingxuan He et al. 'Learning to explore paths for symbolic execution'. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2526–2540.

[104] Wei-Lun Huang and Kang G Shin. 'ES-FUZZ: Improving the Coverage of Firmware Fuzzing with Stateful and Adaptable MMIO Models'. In: *arXiv preprint arXiv:2403.06281* (2024).

[105] Timothy Trippel. 'Developing Trustworthy Hardware with Security-Driven Design and Verification'. PhD thesis. 2021.

[106] Sushma Kalle et al. 'CLIK on PLCs! Attacking control logic with decompilation and virtual PLC'. In: *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*. 2019, pp. 1–12.

[107] Ruimin Sun et al. 'SoK: Attacks on industrial control logic and formal verification-based defenses'. In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 385–402.

[108] Mu Zhang et al. 'Towards automated safety vetting of PLC code in real-world plants'. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 522–538.

[109] Lantao Yu et al. 'Seqgan: Sequence generative adversarial nets with policy gradient'. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 31. 1. 2017, pp. 4681–4690.

[110] Majid Salehi, Danny Hughes and Bruno Crispo. 'μSBS: Static binary sanitization of bare-metal embedded devices for fault observability'. In: *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association. 2020, pp. 381–395.

[111] Shengjian Guo, Meng Wu and Chao Wang. 'Symbolic execution of programmable logic controller code'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 326–336.

[112] Zhicheng Hu et al. 'GANFuzz: a GAN-based industrial network protocol fuzzing framework'. In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 2018, pp. 138–145.

[113] Aoshuang Ye et al. 'Rapidfuzz: Accelerating fuzzing via generative adversarial networks'. In: *Neurocomputing* 460 (2021), pp. 195–204.

[114] Nicole Nichols et al. 'Faster fuzzing: Reinitialization with deep neural models'. In: *arXiv preprint arXiv:1711.02807* (2017).

[115] Shintaro Fujita et al. 'OpenPLC based control system testbed for PLC whitelisting system'. In: *Artificial Life and Robotics* 26 (2021), pp. 149–154.

[116] Naman Govil, Anand Agrawal and Nils Ole Tippenhauer. 'On ladder logic bombs in industrial control systems'. In: *Computer Security: ESORICS 2017 International Workshops, CyberICPS 2017 and SECPRE 2017, Oslo, Norway, September 14-15, 2017, Revised Selected Papers 3*. Springer. 2018, pp. 110–126.

[117] Fabrice Bellard. 'QEMU, a fast and portable dynamic translator.' In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. Califor-nia, USA. 2005, pp. 10–5555.

[118] Antonia Creswell et al. 'Generative adversarial networks: An overview'. In: *IEEE signal processing magazine* 35.1 (2018), pp. 53–65.

[119] DW Pessen. 'Ladder-diagram design for programmable controllers'. In: *Automatica* 25.3 (1989), pp. 407–412.

[120] Jin-woo Myung and Sunghyuck Hong. ‘ICS malware Triton attack and countermeasures’. In: *International Journal of Emerging Multidisciplinary Research (IJEMR)* 3.2 (2019), pp. 13–17.

[121] Zachary H Basnight. ‘Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers’. MA thesis. Graduate School of Engineering and Management, Air Force Institute of Technology, Ohio, 2013.

[122] Brendan Dolan-Gavitt et al. ‘Lava: Large-scale automated vulnerability addition’. In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, pp. 110–121.

[123] Dongdong She et al. ‘Neuzz: Efficient fuzzing with neural program smoothing’. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 803–817.

[124] Eyasu Getahun Chekole et al. ‘Taming the War in Memory: A Resilient Mitigation Strategy Against Memory Safety Attacks in CPS’. In: *arXiv preprint arXiv:1809.07477* (2018).

[125] Yuwei Li et al. ‘UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.’ In: *USENIX Security Symposium*. 2021, pp. 2777–2794.

[126] Yan Shoshitaishvili et al. ‘SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis’. In: *IEEE Symposium on Security and Privacy*. 2016.

[127] Mingzhe Wang et al. ‘Data Coverage for Guided Fuzzing’. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 2511–2526.

[128] Luca Borzacchiello, Emilio Coppa and Camil Demetrescu. ‘FUZZOLIC: Mixing fuzzing and concolic execution’. In: *Computers & Security* 108 (2021), p. 102368.

[129] Sanoop Mallissery and Yu-Sung Wu. ‘Demystify the fuzzing methods: A comprehensive survey’. In: *ACM Computing Surveys* 56.3 (2023), pp. 1–38.

[130] Qiuping Yi, Yifan Yu and Guowei Yang. ‘Compatible Branch Coverage Driven Symbolic Execution for Efficient Bug Finding’. In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024), pp. 1633–1655.

[131] Paolo Felli et al. ‘Data-aware conformance checking with SMT’. In: *Information Systems* 117 (2023), p. 102230.

[132] Cornelius Aschermann et al. ‘REDQUEEN: Fuzzing with Input-to-State Correspondence.’ In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[133] Adrian Herrera, Mathias Payer and Antony L Hosking. 'DatAFLow: Toward a data-flow-guided fuzzer'. In: *ACM Transactions on Software Engineering and Methodology* 32.5 (2023), pp. 1–31.

[134] Gary A Kildall. 'A unified approach to global program optimization'. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, pp. 194–206.

[135] V Aho Alfred, S Lam Monica and D Ullman Jeffrey. *Compilers principles, techniques & tools*. pearson Education, 2007.

[136] Sandra Rapps and Elaine J. Weyuker. 'Selecting software test data using data flow information'. In: *IEEE transactions on software engineering* 4 (1985), pp. 367–375.

[137] Gerard Holzmann et al. 'Static source code checking for user-defined properties'. In: *Proc. IDPT*. Vol. 2. 2002.

[138] Ron Cytron et al. 'Efficiently computing static single assignment form and the control dependence graph'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[139] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2022.

[140] Diego Novillo. 'Tree SSA a new optimization infrastructure for GCC'. In: *Proceedings of the 2003 gCC developers' summit*. 2003, pp. 181–193.

[141] Alessandro Mantovani, Andrea Fioraldi and Davide Balzarotti. 'Fuzzing with data dependency information'. In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2022, pp. 286–302.

[142] Tielei Wang et al. 'TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection'. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 497–512.

[143] Soomin Kim et al. 'Testing intermediate representations for binary analysis'. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 353–364.

[144] Thomas Reps, Susan Horwitz and Mooly Sagiv. 'Precise interprocedural dataflow analysis via graph reachability'. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, pp. 49–61.

[145] Manh-Dung Nguyen et al. 'Binary-level directed fuzzing for use-after-free vulnerabilities'. In: *23rd International Symposium on Research in Attacks, Intrusions and Defences (RAID 2020)*. 2020, pp. 47–62.

[146] Stefan Nagy et al. 'Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing'. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1683–1700.

[147] Jonathan Metzman et al. 'Fuzzbench: an open fuzzer benchmarking platform and service'. In: *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2021, pp. 1393–1403.

[148] Jiashun Wang et al. 'FeedbackFuzz: Fuzzing Processors via Intricate Program Generation with Feedback Engine'. In: *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2025, pp. 1–5.

[149] Matthew Edwin Weingarten, Nora Hossle and Timothy Roscoe. 'High throughput hardware accelerated coresight trace decoding'. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2024, pp. 1–6.

[150] Neil Walkinshaw et al. 'Bounding random test set size with computational learning theory'. In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 2538–2560.

[151] Porfirio Tramontana et al. 'Developing and evaluating objective termination criteria for random testing'. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.3 (2019), pp. 1–52.

[152] 'Trends and Challenges in the Vulnerability Mitigation Landscape'. In: Santa Clara, CA: USENIX Association, Aug. 2019.

[153] Hanhaotian Liu and Tomoharu Ugawa. 'Porting System Software to CHERI: Lessons from Porting CRuby'. In: *日本ソフトウエア科学会第 42 回大会講演論文集*. Proceedings of the 42nd Annual Conference of the Japan Society for Software Science and Technology (JSSST 2025). Unrefereed proceedings. Tokyo, Japan, 3rd–5th Sept. 2025.

[154] Robert NM Watson et al. *Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 7)*. Tech. rep. University of Cambridge, Computer Laboratory, 2019.

[155] William M McKeeman. 'Differential testing for software'. In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[156] Yuting Chen, Ting Su and Zhendong Su. 'Deep differential testing of JVM implementations'. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1257–1268.

[157] Earl T Barr et al. 'The oracle problem in software testing: A survey'. In: *IEEE transactions on software engineering* 41.5 (2014), pp. 507–525.

[158] Yuting Chen et al. 'Coverage-directed differential testing of JVM implementations'. In: *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 85–99.

[159] Nicholas H Tollervey. *Programming with MicroPython: embedded programming with microcontrollers and Python*. " O'Reilly Media, Inc.", 2017.

[160] Duncan Lowther et al. 'Secure Scripting with CHERIoT MicroPython'. In: *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. 2025, pp. 180–191.

[161] Duncan Lowther, Dejice Jacob and Jeremy Singer. 'Morello MicroPython: a python interpreter for CHERI'. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 2023, pp. 62–69.

[162] Andrew G Clark, Neil Walkinshaw and Robert M Hierons. 'Test case generation for agent-based models: A systematic literature review'. In: *Information and Software Technology* 135 (2021), p. 106567.

[163] Michaël Marcozzi et al. 'Compiler fuzzing: How much does it matter?' In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–29.

[164] Xuejun Yang et al. 'Finding and understanding bugs in C compilers'. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.

[165] Mayank Sharma, Pingshi Yu and Alastair F Donaldson. ‘Rustsmith: Random differential compiler testing for rust’. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 1483–1486.

[166] Andrew R Bernat and Barton P Miller. ‘Anywhere, any-time binary instrumentation’. In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. 2011, pp. 9–16.