

HARMFuzz: An efficient QEMU-Assisted Hybrid ARM Fuzzer

Bachelor thesis by Alisher Darmenov (Student ID: 2475840)

Date of submission: May 16, 2025

1. Review: Prof. Dr.-Ing. Ahmad-Reza Sadeghi
2. Review: M. Sc. Mohamadreza Rostami
Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
System Security Lab

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Alisher Darmanov, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

English translation for information purposes only:

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Alisher Darmanov, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt without any outside support and using only the quoted literature and other sources. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I have clearly marked and separately listed in the text the literature used literally or in terms of content and all other sources I used for the preparation of this academic work. This also applies to sources or aids from the Internet.

This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Datum / Date:

16.05.2025

Unterschrift/Signature:



Contents

1. Introduction	6
2. Preliminary	11
2.1. Fuzzing	11
2.1.1. Coverage	12
2.1.2. Mutation	14
2.1.3. Corpus	15
2.2. Hardware Fuzzing	16
2.2.1. CPU Fuzzing Types	17
2.2.2. Hardware Coverage	17
2.3. ELF File Structure	18
2.4. QEMU and QEMU KVM	19
2.4.1. QEMU	19
2.4.2. QEMU KVM	19
3. Related Works	20
4. HARMFuzz Design	24
4.1. Overview of HARMFuzz	24
4.1.1. Key Contributions	24
4.1.2. HARMFuzz Architecture	25
4.2. HARMFuzz Framework	26
4.3. Creation of Test Input	27
4.4. Coverage Collection	29
4.5. Design Considerations	30
5. Implementations	33
5.1. Software	33
5.1.1. Mutation Implementation of HARMFuzz	34



5.1.2. Coverage Collection	36
5.1.3. QEMU and QEMU-KVM Configuration Run	37
5.2. Hardware	38
5.2.1. Raspberry Pi 4 Model B	39
6. Performance Evaluation	41
7. Discussion	46
7.1. Advantages	46
7.2. Limitations	47
8. Conclusion	48
8.1. Acknowledgment	49
A. Appendix Developed Mutator	50

Abstract

ARM architectures are central to modern devices, making their security critical. Fuzzing has shown promising results in detecting vulnerabilities; however, traditional fuzzing tools focus on software vulnerabilities and are not designed to interact directly with hardware or emulate processor-specific behaviors, leaving a critical gap in detecting flaws unique to hardware implementations.

The core question of this thesis is how differential fuzzing can be used to identify vulnerabilities in ARM processors, considering their hardware complexity, by integrating emulated environments with hardware testing.

This thesis introduces HARMFuzz, an efficient ARM-specific black-box hardware fuzzing tool designed to overcome the limitations of traditional fuzzers, such as AFL++ when testing ARM processors, primarily such fuzzers focus on software vulnerabilities and struggle with hardware-specific challenges. HARMFuzz solves this problem by eliminating trimming and employing dynamically controlled mutation without changing the file size.

Another issue with traditional fuzzers is their limited hardware awareness, due to hardware complexity and closed internal processor code. This makes black-box fuzzing focus on the overall behavior of the processor, which complicates internal processor discovery. The developed fuzzer can detect processor-specific bugs without requiring full internal knowledge through the integration of QEMU and KVM.

By systematically addressing these limitations, HARMFuzz provides a solution for improving the effectiveness of the vulnerability discovery process in ARM hardware by achieving higher code coverage and identifying 1,000 additional edges in QEMU-only environments and 100 in KVM-enabled scenarios compared to the state-of-the-art works. The study highlights its impact on enhancing ARM chip security and demonstrates promising potential for further increasing adaptability and efficiency in future research.

1. Introduction

Today, ARM chips are the most common chips in the world [43]. In the fourth quarter of 2020, more than 6.7 billion ARM-based chips were produced, surpassing the combined production of x86, ARC [42], Power [50], and MIPS-powered chips [33]. Due to their low cost, low power consumption, and low heat generation, ARM chips are used almost everywhere, from smart bulbs to embedded systems, as well as in nearly all smart devices such as smartphones and smart home technologies [43].

The widespread of ARM processors in the community makes the question of CPU Security more important each day. Hackers have a much bigger attack surface and now they can steal not only process personal and sensitive data, for example, in 2022, a Korean hacker was arrested for accessing and distributing private videos from vulnerable wall-pad cameras in over 400,000 homes [9], but also interfere in people's physical security, for example, open smart locks on doors to criminals[39].

In recent years, several critical CPU vulnerabilities have been discovered [53] [15] [46] [32] [31] [26], revealing architectural flaws in widely deployed processors. These vulnerabilities, often rooted deep within microarchitectural behavior, demonstrate that even thoroughly tested CPUs can harbor subtle bugs that remain undetected for years.

For example, Spectre [53] and Meltdown [15] broke the fundamental assumptions of memory isolation. Spectre tricks speculative execution mechanisms into leaking data across application boundaries, while Meltdown allows user-space processes to read privileged kernel memory. These attacks affected processors from Intel, AMD, and ARM, exposing billions of devices.

Beyond these, newer classes of vulnerabilities have continued to emerge. Foreshadow (L1 Terminal Fault) [46] exploited Intel's implementation of secure enclaves Software Guard Extensions (SGX), bypassing the guarantees of Trusted Execution Environments (TEEs) by leaking data from L1 cache. Similarly, ZombieLoad [32] and RIDL [31] demonstrated

how Microarchitectural Data Sampling (MDS) attacks can retrieve data across security boundaries using internal CPU buffers.

More recently, Zenbleed [26], a vulnerability specific to AMD’s Zen 2 architecture, was discovered by Tavis Ormandy from Google Project Zero. It allows the leakage of sensitive register content across threads by exploiting speculative execution and register renaming mechanisms. Unlike earlier vulnerabilities, Zenbleed could be triggered without specific system calls, making it a potent example of hardware-level leakage that bypasses typical privilege boundaries.

Even after these vulnerabilities were discovered, addressing them via software patches remains exceptionally difficult. Most CPU bugs are ingrained in the CPU’s setup, and there are no simple fixes. This underscores the need for more robust hardware-level solutions to tackle these complex security issues. For example, mitigating the Meltdown vulnerability [15], required a dramatic separation of the kernel’s memory from user processes, which slowed down the performance of the entire system [8].

It is important to develop testing techniques that can discover and resolve CPU bugs and vulnerabilities faster than hackers can exploit them.

To achieve high-security standards, chip manufacturers carefully test each model before mass production. However, as processor complexity continues to increase each year in line with Moore’s law [52], traditional testing methods are no longer sufficient to detect deeper and more complex vulnerabilities that could compromise the security of manufactured chips. Fortunately, a technique with higher potential than standard methods has emerged in this field: hardware fuzzing, automated tool for hardware testing by guided mutations [30].

Several hardware fuzzers have been developed to test CPU behavior and detect hardware-level vulnerabilities. For example, Silifuzz [21] is a large-scale fuzzer designed for detecting CPU misbehavior in x86 processors using emulation and differential execution. However, it is tightly coupled with the x86 architecture and does not support ARM or other ISAs. Armshaker [36], on the other hand, targets the Armv8-A architecture and focuses specifically on identifying undocumented or hidden instructions. While useful, its detection scope is narrow and does not cover broader behavioral anomalies. RISCvuzz [38] is a fuzzer tailored for the RISC-V architecture that compares execution results between real processors to detect inconsistencies. This approach, however, requires access to multiple physical RISC-V implementations and relies on direct hardware coverage collection, which can be difficult to fulfill in practice.

Although each of these tools provides valuable insights, they are limited either by architecture specificity, a narrow detection focus, or practical constraints such as hardware availability. This highlights the need for an automated and architecture-aware testing framework capable of uncovering complex and subtle instruction sequences that lead to abnormal behavior—particularly in widely deployed ARM processors.

To address these limitations and bridge the gap between existing approaches, this thesis introduces a ARM fuzzer, HARMFuzz—an automated testing tool designed to enhance hardware fuzzing efficiency while avoiding the complexity of hardware coverage collection on post-manufactured ARM processors. HARMFuzz employs differential fuzzing techniques to automatically generate and execute diverse instruction sequences, enabling the discovery of subtle hardware misbehaviors without requiring access to physical hardware internals.

HARMFuzz performs differential fuzzing by comparing the register states between two execution environments: a **Fully Emulated Environment**, where an ARM processor is emulated entirely in software using QEMU [28], and a **Hybrid Environment**, where QEMU is combined with KVM virtualization to run code natively on the host’s ARM hardware.

By treating the emulated processor as software under test, HARMFuzz significantly simplifies the hardware fuzzing process. It avoids the complexities and limitations of direct hardware instrumentation, making fuzzing more accessible, scalable, and efficient. This software-based approach enables automated discovery of subtle differences in execution behavior, helping to uncover potential hardware-level flaws without requiring access to physical test platforms or custom instrumentation.

Our goals and contributions. The primary goal of this thesis is to develop an efficient and automated fuzzing tool specifically tailored for ARM processors, capable of detecting hardware-specific vulnerabilities that are difficult to identify using traditional software-focused fuzzing techniques.

Several challenges made this process particularly difficult. First, collecting coverage from hardware is a complex task. ARM chips are closed source, meaning that internal code and interactions cannot be directly accessed or processed, making coverage-guided fuzzing both challenging and nearly impossible. Secondly, software fuzzers cannot be directly applied to structured files such as ELF files, which are used as test cases for hardware fuzzers, due to the strict structure of such files. Any damage or inconsistency in these files leads to crashes that are unrelated to security flaw exploration, thereby decreasing the effectiveness of these approaches.


Additionally, generating good fuzzing seeds (the initial test inputs) for hardware is a challenge. For proper fuzzing it is necessary to manually compile and collect unique test cases that will trigger different path exploration inside of CPU and can be used for more efficient guided mutations, the proper amount of seeds could be essential for efficient fuzzing process and faster coverage increase.

Finally, directly fuzzing hardware is a slow process. Hardware fuzzing requires significant time from boot-up to completion on a real machine. Additionally, even emulation can be extremely slow, particularly when using weaker processors, making the testing process more time-consuming and less efficient.

To solve these challenges, HARMFuzz uses QEMU as a fuzzing proxy that emulates similar, though not identical, behavior to a real CPU. This approach allows fuzzing of emulated hardware with software-based fuzzing techniques, while performing differential fuzzing by comparing results with a QEMU KVM run, which uses the actual CPU and enables the collection of register traces without the need for specialized hardware. Additionally, to maintain the integrity of binary files such as ELF files, HARMFuzz employs custom mutators based on AFL++ to prevent trimming and preserve the strict structure of the binary. The generation of interesting test cases can be efficiently carried out on a high-performance machine through QEMU emulation, and then these test cases can be used in the QEMU KVM environment for comparison. This approach significantly enhances the fuzzing process, making it more efficient and less complex.

In summary, the contributions of this thesis represent a significant advancement in improving ARM processor security through more efficient automated fuzzing approach. The introduction of HARMFuzz addresses key challenges posed by the increasing complexity of ARM architectures and the limitations of other fuzzing approaches, enabling more efficient and effective hardware fuzzing process.

The thesis is structured as follows: First, the background of key concepts §2, file structures, and techniques relevant to hardware fuzzing will be introduced. Next, the related works Section §3 reviews existing tools and academic research in hardware fuzzing, comparing their capabilities to those of HARMFuzz and highlighting its innovative aspects. The design of HARMFuzz in Section §4 will then be detailed, covering the creation of ELF files used as input and the solutions proposed to overcome hardware fuzzing challenges. The implementation Section §5 outlines the software and hardware setup used in the experimental phase, including mutation techniques and execution setups. Following this, the evaluation Section §6 presents experimental results comparing HARMFuzz's performance with AFL++, demonstrating its efficiency in detecting CPU misbehaviors. The discussion Section §7 summarizes the design and evaluation outcomes, highlighting



both the strengths and limitations of HARMFuzz. Finally, the thesis concludes with a summary of findings and suggestions for future work Section §8.

2. Preliminary

Fuzzing has developed into a sophisticated area of research characterized by a diverse set of specialized terms and concepts. To facilitate a clear understanding of HARMFuzz, it is essential to first establish the foundational terminology commonly used in fuzzing frameworks. This section provides an overview of these terms and presents the necessary background to comprehend the design and functioning of HARMFuzz.

2.1. Fuzzing

In the context of programming and software engineering, fuzzing—also known as fuzz testing—is an automated technique used to detect bugs, vulnerabilities and abnormal behavior in software by supplying it with malformed, unexpected, or random input data. The software’s behavior is then observed for signs of failure, such as crashes, assertion violations, or memory issues [16]. Smart Fuzzers are particularly effective when applied to programs that process structured inputs, where the expected input format is predefined, such as a specific file type or communication protocol [11]. However, recent studies suggest that fuzzing structured inputs remains challenging and often requires structure-aware approaches to be effective [19]. In comparison, unstructured or “dumb” fuzzers generate random inputs, which frequently result in invalid or malformed files that fail to pass even the initial parsing stages [49]. A well-designed smart fuzzer, by contrast, generates inputs that are partially valid: they are structured enough to bypass initial checks, yet still malformed enough to expose edge cases or overlooked bugs deeper in the program logic [49].

From a security perspective, the most critical inputs to fuzz are those originating from untrusted sources—such as user input, uploaded files, or network data—especially when they cross trust boundaries (i.e., when external input interacts with privileged or sensitive internal components) [49]. For example, fuzzing code that handles user-uploaded files

is typically more valuable than targeting components that only process trusted internal configuration files [49].

One common approach to distinguishing fuzzers uses the terms white/grey and black box fuzzing. White-box fuzzers have complete access to a program's internal structure, using different analysis techniques to generate optimized inputs to cover the program well, though at the cost of increased computational expense [17]. In contrast, Black-box fuzzers test a program without any knowledge of its internal structure, relying on random inputs for fast execution, though this often leads to poor coverage [17]. Grey-box fuzzers balance speed and coverage by analyzing program execution to select inputs that cover new code, while avoiding the computational cost of deep analysis [17].

There are two main types of fuzz testing: coverage-guided and behavioral. Modern Fuzzing techniques based on coverage-guided mutations. These approaches aim to maximize code coverage and improve the discovery of hidden bugs by analyzing program execution paths during fuzzing [2]. The other type is Behavioral or random fuzz testing compares an application's actual behavior with its expected behavior, using random inputs to identify inconsistencies, which often reveal bugs or security risks [2].

To detect bugs, fuzzers rely on methods that distinguish normal from abnormal program behavior, despite the challenge known as the test oracle problem [49], where machines can't always distinguish a bug from a feature. One common approach is crash detection, where the fuzzer identifies inputs that cause a program to crash, such as a buffer overflow in a C program that leads to undefined behavior. To catch more issues, sanitizers are used to monitor runtime behavior and force a crash when they detect errors like memory leaks [6], use-after-free [5], or race conditions [7] —for example, AddressSanitizer [5] can detect a hidden buffer overflow even if it doesn't cause an immediate crash. Another effective technique is differential testing, where the same input is run on two versions or implementations of a program, and any differences in output suggest a potential bug. These methods help fuzzers uncover both obvious and hidden vulnerabilities in software [49].

2.1.1. Coverage

The data coverage is a crucial part of the fuzzing process[41]; it is used as a metric for grading the success of a fuzzing iteration. There are plenty of metrics that could be used as coverage data. Here will be presented only three types of coverage data that are relevant to the current thesis.

```
1 void describe_number(int number) {
2     if (number % 2 == 0) {
3         printf("%d is even.\n", number);
4     } else {
5         printf("%d is odd.\n", number);
6     }
7 }
8
9 int main() {
10     int num = 5;
11     describe_number(num);
12     return 0;
13 }
```

Figure 2.1.: Example of code

The first one, **Code Coverage**, is the metric that measures the number of triggered unique edges(transitions between basic blocks), basic blocks, or functions during the program run. A basic block is a straight-line sequence of code with no branches, containing only one entry point at the beginning and one exit point at the end [44]. During each execution in testing software, the fuzzer collects these data.

For example, in Figure 2.1 and 2.2, the function **main** serves as the entry point of the program and calls **describe_number** with a sample input. This execution starts at Basic Block 1 (line 9,10), where the variable is initialized and the function is called. The control then transitions into the corresponding basic blocks within **describe_number**, depending on whether the number is even or odd. After the function call returns, Basic Block END (line 12) is triggered, which completes the program with a return statement. However, the code itself cannot provide information about how many edges were used during its execution. To solve this problem, special code coverage tools and instruments are used, such as *gcov* [13], *LLVM's Sanitizers* [37], and *coverage.py* [1].

A recent software coverage-related approach that utilizes the Program Counter (PC) as an analysis metric was introduced in [25], setting itself apart from the traditional coverage models employed by modern fuzzers. The program counter, a special-purpose processor register, holds the address of the next instruction to be executed. By analyzing patterns in PC values during execution, this method enables fine-grained insight into program flow and memory usage, offering the potential to detect subtle issues such as memory leakage. Unlike conventional fuzzing tools, which use coverage metrics primarily to guide test input generation, this technique leverages PC behavior in a security-oriented context

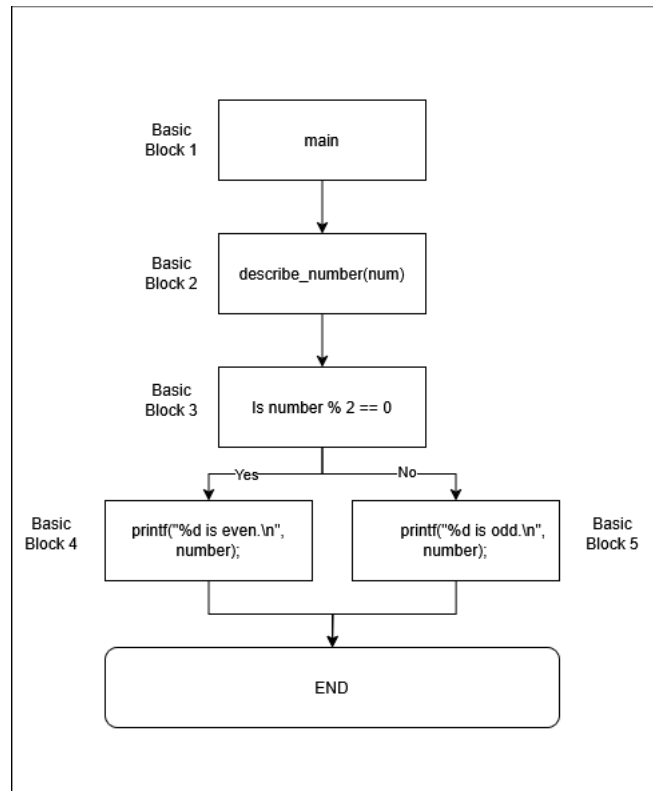


Figure 2.2.: The Basic Block Diagram of 2.1

to reveal side-channel vulnerabilities.

2.1.2. Mutation

A mutator is a part of a fuzzer that takes one or more inputs and changes them to create new inputs, often using simple rules like flipping bits (0 becomes 1, and 1 becomes 0) or replacing bytes [12]. For example, Coverage-based mutators modify input files according to coverage data, with the goal of constantly changing input files using different mutation strategies to achieve better coverage results [55].

There are also mutators that operate without coverage feedback, relying instead on random, blind modifications [55]. Regardless of the approach, the core responsibility of a

mutator is to systematically explore the input space in a way that maximizes the chances of discovering bugs or reaching deeper program logic [55].

The mutation process ranges from simple bit flips, where random bits are flipped (0 becomes 1, and 1 becomes 0), to more complex mutations, such as grammar-aware mutations [35], which use knowledge of input structure or grammar to generate syntactically valid but semantically unexpected inputs. However, the most commonly used approach is combination mutations [24], which apply different mutation techniques. This helps create more diverse test cases that trigger higher coverage and make the fuzzing process more efficient.

For example, in Figure 2.1 and 2.2, when `num = 5` is used in the execution, Basic Block 5 is reached, but Basic Block 4 is not. After collecting this coverage data, the mutator applies a mutation strategy, such as bit-flipping, to the current input. In this case, the binary representation of `num = 5` is `0101`. By flipping a bit, the value changes to `num = 4`, whose binary representation is `0100`. This mutation creates a new input that triggers a different coverage path, potentially leading to new parts of the program being exercised, such as Basic Block 4.

2.1.3. Corpus

A corpus refers to the initial set of input data used by the fuzzer, which is typically collected by the user to begin the fuzzing process on the System Under Test (SUT). The corpus serves as the starting point for generating mutated inputs during fuzzing.

A well-constructed corpus is essential for the effectiveness of fuzzing, as it determines the variety of inputs that the fuzzer will work with. Ideally, the corpus should contain a diverse range of valid and relevant inputs that produce diverse coverage results. This diversity enables the fuzzer to explore a broader set of execution paths, improving the likelihood of discovering vulnerabilities or unexpected behaviors.

Corpus data can be collected in various ways, such as manually gathering example files or inputs, using existing datasets, or even extracting data from logs or real-world user interactions. For instance, when fuzzing a file parser, the corpus could consist of different file formats, such as PDFs, images, or text files, each representing a valid example that the software might encounter. In some cases, it may be beneficial to include malformed or corrupted files in the corpus to help identify handling issues for invalid inputs.

The quality of the corpus directly affects the success of fuzz testing. A larger and more varied corpus increases the chances of exposing edge cases and flaws in the software. However, a poorly chosen corpus, with too few examples or lacking diversity, may limit the fuzzer’s ability to discover critical issues. One example of a publicly available corpus is the set of “seed” inputs used in the American Fuzzy Lop (AFL++) fuzzing tool, which has been shown to help achieve high code coverage in various programs[24].

2.2. Hardware Fuzzing

Existing approaches for detecting hardware vulnerabilities include manual testing, formal verification, and benchmarking. While each method contributes to identifying flaws, they also suffer from notable limitations in terms of cost, scalability, and the ability to discover unknown bugs.

Fuzzing has been used for decades to test software. The core idea involves feeding the system with a large number of unexpected or malformed inputs to trigger crashes or abnormal behavior [48]. Recently, this technique has been adapted for hardware, with the goal of identifying design vulnerabilities and logic bugs using similar principles. Compared to other testing techniques, fuzzing offers significant advantages in terms of automation, scalability, and its ability to discover previously unknown issues[40][34] [3] [21][38] [14] [27][10][22][36]. The most obvious testing technique is manual testing, where a person manually examines the entire hardware design [51]. But it requires many experts, time, and money. This approach is inefficient for hardware designs.

Another approach is formal verification, which checks whether the model works correctly using formal mathematics [47]. This method is better than manual testing, but it is still expensive, requires much expert knowledge, and is slow. Also, due to the state explosion problem—the extremely large number of possible states—it cannot be effectively used for complex hardware designs [20].

A better approach that can be adapted for hardware is benchmarking, which checks the system for known vulnerabilities using known exploits [23]. It does not require many experts, as it focuses on detecting already known bugs. This makes the approach both good and bad: good because it is cheap, fast, and easy to apply; bad because it cannot find new, undiscovered bugs or already known bugs triggered by different, unknown exploits [23]. This makes the approach too limited.

Overall, hardware fuzzing outperforms these testing techniques in discovering hardware bugs. It is automated, and once started, it requires minimal human attention. This approach is much faster than manual testing and does not face scalability issues like formal verification. Thanks to its flexibility in changing inputs, it can detect unknown bugs and exploits [30].

Certainly, hardware fuzzing cannot prove the absence of bugs or vulnerabilities, but its advantages make it an effective tool for testing modern systems [30].

2.2.1. CPU Fuzzing Types

There are two different types of CPU Fuzzing that are based on internal specification availability. One of them **White-Box Fuzzing**, is used when full access to hardware specifications, RTL models, or source code is available. It enables precise testing by analyzing internal states, transitions, and execution flows. Techniques like differential fuzzing, symbolic execution, and guided mutation help detect logic errors and design flaws.

The other is **Black-Box Fuzzing**, in contrast, works without internal hardware knowledge. It tests CPUs by providing inputs, observing outputs, and detecting unexpected behavior. This approach is useful for proprietary architectures, finding undocumented instructions, and identifying hardware misconfigurations.

2.2.2. Hardware Coverage

Code coverage is mostly used as a metric for software testing, but there are other types of coverage that are also used for hardware testing. Collecting hardware coverage data is not a simple task [40]. Specifically, it is not enough to install some specialized software because the data must be collected starting from the boot process, which is a complex task [40]. To measure hardware coverage, register coverage is used (register state tracing) [4]. During execution, all processor registers or only specific ones (such as Control and Status Registers) [4] are monitored, and information is collected on which registers changed and how many were influenced by execution. This data is used for mutation and evaluation in hardware fuzzing.

2.3. ELF File Structure

ELF (Executable and Linkable Format) is a structured file format commonly used for executables, object code, shared libraries, and core dumps. It provides a flexible and organized way to define executable programs, including their code, data, and metadata. In some cases, ELF files can function as bare-metal applications—programs that boot directly on hardware without requiring an operating system—although this requires special compiler setups and additional tools. Many ELF files are designed with the assumption that an operating system will handle tasks like dynamic linking and hardware initialization, meaning they are generally unsuitable for direct hardware boot without modification. This unique property of ELF files makes them a suitable corpus for HARMFuzz, enabling the fuzzing of ARM architecture in an emulated environment.

As shown in Figure 2.3, the ELF File could be divided into six main sections, each section contains necessary data for executing the file [45].

ELF Header is the section that contains metadata that assists in identifying ELF type, specifying an architecture, and information about the ELF Header's size, Program, and Section Headers' Offset, and other data that is readable for proper interpretation and loading of the file into memory. The **Program Header** Table defines which segments of an ELF file are loaded into memory when executed. It usually includes loadable sections such as `.text`, `.data`, and `.bss`. The header specifies details like file size, memory size, virtual and physical addresses, as well as access permissions (e.g., read/write). The code section (`.text` section) is part of the ELF File containing executable code. It is a read-only and executable section. The data section (`.data` section) is the part that contains initialized global and static variables.

There are other sections that could be included in the ELF File, but we need a `.bss` section. This section contains Uninitialized Variables that were declared but not initialized. This

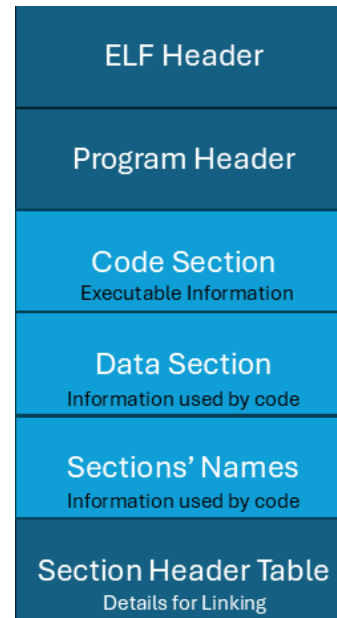


Figure 2.3.: ELF file structure

section is writable due to this, it can be actualized during the execution of the ELF File. This section is also responsible for memory allocation at load time.

The last section is **Section Header** Table; this table contains the sections' index, name, flags, addresses, and each section offset.

2.4. QEMU and QEMU KVM

2.4.1. QEMU

QEMU [28] (Quick Emulator) is an open-source emulator and virtualizer capable of emulating an entire hardware system, from the CPU to peripheral devices. It enables the execution of software designed for one hardware platform on a different platform by translating guest instructions into host instructions. However, this translation process incurs a performance overhead, making QEMU's software virtualization generally slower than running directly on native hardware.

2.4.2. QEMU KVM

QEMU KVM [28] integrates the standard QEMU emulator with **KVM** (Kernel-Based Virtual Machine), a Linux kernel module that transforms the Linux kernel into a bare-metal hypervisor. This integration enables virtual machines to run directly on the host CPU with minimal overhead. By combining QEMU's hardware emulation capabilities with KVM's efficient CPU virtualization, QEMU KVM leverages the real CPU while still emulating peripheral hardware components.

In standard QEMU, instructions are translated and executed by an internal interpreter. However, in QEMU KVM, guest instructions are executed directly on the physical CPU without translation, which leads to significantly better performance. As a result, offloading CPU tasks to KVM allows QEMU KVM to achieve the behavior of a real CPU, rather than an emulated one.

3. Related Works

CPU fuzzing is a technique for discovering hardware vulnerabilities. As mentioned earlier, it falls into two main categories: **White-Box Hardware Fuzzing** and **Black-Box Hardware Fuzzing**, depending on the availability of internal design details.

This section provides an overview of existing work in the area of hardware fuzzing, particularly for CPU components, and is divided into white-box and black-box approaches. It highlights key methods and tools used in the field, discussing their strengths and limitations. Finally, it positions **HARMFuzz** within this landscape, emphasizing its advantages and novel contributions compared to prior work.

White-Box Hardware Fuzzing refers to testing techniques applied to hardware where the tester has full access to the internal design, including documentation and source models such as RTL (Register-Transfer Level) or HDL (Hardware Description Language). This allows the tester to directly target and explore specific parts of the hardware that might be vulnerable. In differential fuzzing, the same input is provided to multiple hardware models or implementations. These models may differ in design, configuration, or even in the underlying architecture. By comparing how each model behaves when processing the same input, the tester can identify abnormalities or unexpected behavior. If one model crashes or produces incorrect results while others do not, it suggests a potential vulnerability or bug in the faulty model. The key idea is that differences in execution across models help highlight weaknesses that may not be apparent when testing just one model[40].

There are many representative White-Box fuzzers; one of the newest one is *Cascade* [34], the novelty of *Cascade* lies in its ability to generate test programs guided by both data flow and control flow analysis. This allows it to create inputs that explore meaningful execution paths in the hardware design. *Cascade* then executes these RISC-V programs both in simulation (e.g., RTL or a golden model) and on the CPU under test, comparing their behaviors. This dual approach helps find bugs or vulnerabilities that might only

appear in one environment but not the other, making Cascade more effective at detecting hardware issues.

Another work is *Fuzzing Hardware Like Software* [40], which involves converting the RTL hardware model into a software representation using the Verilator simulator. This approach eliminates the need for a physical CPU under test. Instead, it compiles the hardware model into a simulator, which is then fuzzed with a software fuzzer, using traditional software coverage metrics. This method allows for the evaluation of potential hardware vulnerabilities through software fuzzing techniques without requiring actual hardware.

Finally, *WhisperFuzz* [3] is a white-box fuzzer that detects and locates timing vulnerabilities in processors by combining fuzzing with static analysis. Unlike black-box and gray-box fuzzers, it identifies the root causes of timing issues and evaluates timing coverage using microarchitectural state transitions. *WhisperFuzz* instruments processor RTL designs to track execution timing and detect timing leaks, improving vulnerability detection and localization. Finally, *TheHuzz* [18] fuzzes processors using assembly-level instructions based on golden-reference models, high-level simulation or specification of the hardware's expected behavior, to discover software-exploitable hardware bugs. Unlike other hardware fuzzers, it supports commonly used hardware description languages (HDLs) and captures various hardware behaviors, including signal transitions and floating wires.

Black-Box Hardware Fuzzing, in contrast, is applied to architectures with incomplete or inaccessible internal design documentation like x86_64 and ARM, meaning that the fuzzing process relies solely on observing input-output behavior without knowledge of the hardware's internal states or design specifications. The representative example is *SiliFuzz* [21], which is a tool designed to detect CPU bugs and vulnerabilities in x86_64 processors. It emulates CPU behavior using the UNICORN [29] CPU emulator, allowing it to test various inputs and generate predicted outputs safely. These test cases are then run on real hardware, where the actual outputs are compared to the predicted ones. Any difference between the two indicates unexpected or incorrect behavior. This approach enables it to identify different CPU defects, including logic bugs, invalid CPU behaviors, and instances of Silent Data Corruption—problems that do not cause immediate failures but result in incorrect computations. In contrast, *RISCVuzz* [38], employs black-box fuzzing by targeting RISC-V architectures. Unlike *SiliFuzz*, *RISCVuzz* does not use CPU emulation but directly executes the same instruction set across multiple real RISC-V processors. Inconsistencies between various RISC-V implementations with the same input indicates potential vulnerabilities.

On the other hand, *EM-Fuzz* [14] monitors memory interactions within processors during the fuzzing of an emulated processor, making it effective for discovering memory-based vulnerabilities such as buffer overflows. It was tested on ARM32 and x86 architectures using black-box fuzzing techniques. *Hyperpom* [27] is a mutation-based, coverage-guided fuzzer specifically designed for fuzzing 64-bit ARM binaries. It executes programs in an emulated CPU environment using Apple Silicon’s hypervisor, which runs the binaries directly on the host ARM CPU with native behavior, unlike traditional emulators. The mutations applied to inputs are not purely random but are guided by runtime feedback collected during execution—such as which instructions were executed or which memory regions were accessed. The information gathered can be of different types (addresses, stack frames generated, etc.), and so do gathering methods (hardware mechanisms, hooks added at compile-time/runtime, etc.). Finally, there are approaches like *Sandsifter* [10] and *UISFuzz* [22]. The first one finds hidden instructions and hardware bugs in x86 processors by systematically generating machine code to search through a processor’s instruction set and monitoring for abnormal behavior. The last one can be used to find undocumented instruction. Extra, *Armshaker*[36] is fuzzer with focus on finding hidden instruction in ARM chips using emulators, it uses brute force approach to detect any not documented instructions in Armv8-A ISA.

Our proposed **HARMFuzz** focuses on ARM processors’ black-box fuzzing, using QEMU virtualization to emulate real hardware and compare it with real CPU work. This combination of emulation and real hardware interaction allows it to detect a wide range of CPU misbehaviors, such as logic bugs and buffer overflows. Unlike White-Box approaches like Cascade, WhisperFuzz and Fuzzing Hardware Like Software, which depend on complete access to internal design models and RTL simulations, the HARMFuzz operates without such insights. Its differential fuzzing strategy is comparable to that of SiliFuzz, where the behavior of ARM CPUs in virtualized environments is directly compared with real hardware to detect deviations. HARMFuzz is not restricted to emulated environments like EM-Fuzz. Also HARMFuzz can generate inputs on a high-performance machine and then apply them directly to the Device Under Test (DUT), even if the DUT itself has lower performance. This capability allows HARMFuzz to offload input generation and fuzzing tasks to more powerful machines, providing better performance and scalability. In contrast, Hyperpom is restricted to generating and applying inputs directly on the DUT, limiting its overall performance and the ability to leverage external computational resources. By using differential hardware fuzzing, HARMFuzz gains the advantage of interacting with real CPU hardware, making it capable of catching hardware-specific bugs that would be difficult to detect in purely emulated systems. In contrast to highly specialized fuzzers like BaseSAFE, which targets particular vulnerabilities (e.g., side-channel attacks), or

Sandsifter and Armshaker, which focus on hidden instructions, HARMFuzz’s strength lies in its wide-range detection of CPU anomalies. Additionally, in contrast to RISCVue [38], HARMFuzz is not demanding more real CPUs for comparison, but the whole fuzzing process is processed on top of the same CPU on the same hardware. Such an approach allows it to comprehensively validate the consistency of processor behavior across both real and virtualized environments, providing a realistic and thorough evaluation of ARM CPU functionality in modern hardware configurations.

4. HARMFuzz Design

This chapter presents the architecture and design principles of HARMFuzz, a differential fuzzing framework targeting ARM processors. Each part of HARMFuzz is described in details in the subsequent sections. Finally, the major design challenges and the corresponding solutions are discussed in the Implementation section (§ 5).

4.1. Overview of HARMFuzz

HARMFuzz is an framework for an automated test of ARM processors that uses a differential fuzzing approach, it aims to detect differences between the behavior of real ARM hardware and its emulated processor. The system generates test inputs in the form of ELF binaries, executes them on both the physical Device Under Test (DUT) and an emulated ARM environment, and then compares the resulting register states and coverage data. Any abnormalities between the two execution traces may indicate the presence of hardware bugs or undocumented instructions.

4.1.1. Key Contributions

HARMFuzz introduces the following key contributions:

1. A mutation-enabled ELF input generation engine designed specifically for bare-metal ARM systems.
2. A hybrid fuzzing loop combining high-speed emulation and real-device testing.
3. The approach to extract hardware coverage with minimal complexity.
4. A framework that is scalable and portable to different ARM boards.

4.1.2. HARMFuzz Architecture

The architecture of HARMFuzz consists of two phases: **Corpora Generation** and **Bug Detection**, as illustrated in Figure 4.1. This modular structure enables the system to scale and adapt to different test scenarios involving real and emulated ARM processors.

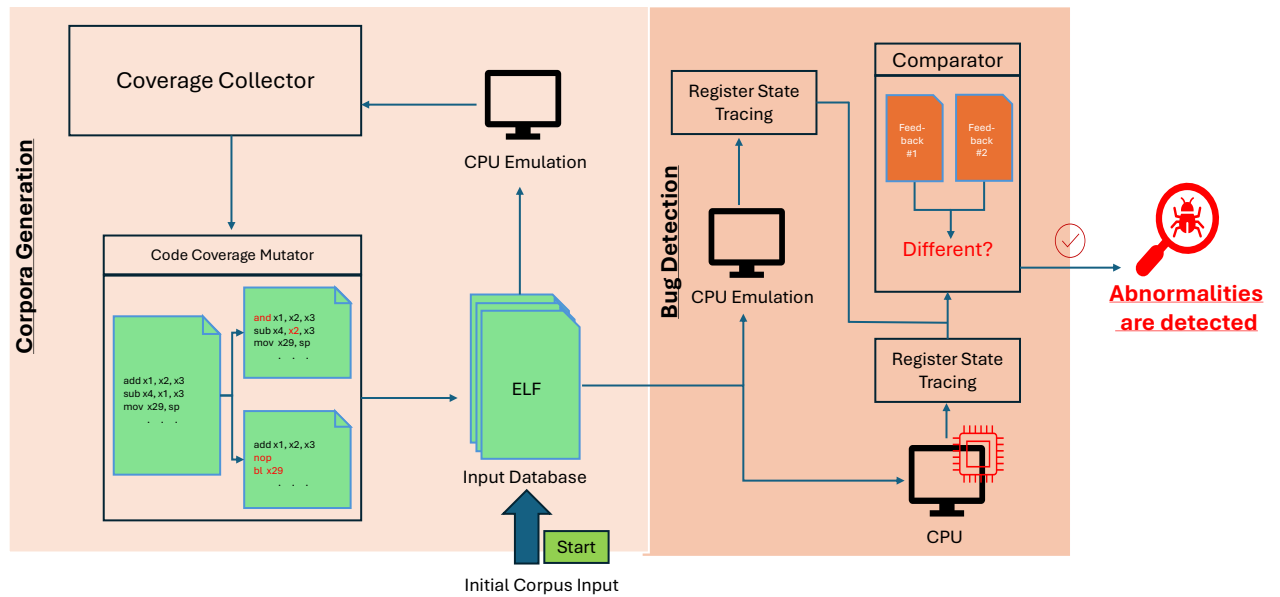


Figure 4.1.: The main design of HARMFuzz

Corpora Generation.Corpora Generation. This phase begins with an initial set of test cases (ELF files), which serve as input to the *CPU Emulation*. Each mutated ELF file is executed in an emulated ARM CPU environment, and its execution trace is analyzed by the *Coverage Collector* to assess code coverage. The HARMFuzz mutator applies various combinations of mutations to generate new test inputs, but restricts them only to the main section of the ELF file. This ensures that structural elements such as headers and memory layout remain unchanged, preserving syntactic correctness and preventing boot failures. Inputs that lead to increased, novel coverage, triggered crashes or timeouts are

stored in the input database for further testing.

Bug Detection. After generating a sufficient set of test cases, HARMFuzz starts the differential analysis. In this stage, each ELF file is executed on both the emulated ARM CPU and the physical Processor Under Test (PUT). During execution, the *Register State Tracing* module records the values of general-purpose and system registers after each instruction. These execution traces are subsequently analyzed by the *Comparator*, which performs a one-to-one comparison between the emulated and physical traces. Any differences in register states may signify undocumented processor behavior, hardware anomalies, or inaccuracies in the emulation model. Such abnormalities are used for further investigation.

4.2. HARMFuzz Framework

HARMFuzz is based on the principles of differential fuzzing. It compares the registers states between CPU and emulated CPU, any difference between tracings could potentially indicate CPU bug or vulnerability.

It consists of two main steps: Corpora Generation and Bug Detection. Corpora generation is important part that can generate corpora in high performance platform and then test them on platform with less performance features, more specifically:

1. Collected ELF files are used as corpus input for HARMFuzz, generating new ELF files.
2. ELF files are used as kernels in an emulated ARM hardware environment, replicating the execution conditions of the DUT.
3. During kernel execution, coverage data is collected and used by the Coverage-based Mutator to generate new ELF files.
4. Test cases that trigger higher coverage are saved for future tests on real ARM hardware. Along with the test cases, their register coverage is also stored for comparison in the next step.

The generation of ELF files can be performed in a high-performance environment, which is faster than the Device Under Test, allowing for generating more new corpora.

Once enough input and feedback file sets have been collected, HARMFuzz uses them for bug and vulnerability detection.

The collected and generated kernels, along with the feedback, are manually or automatically tested one by one on the real Device Under Test.

1. Each ELF file will be used as a kernel on the real Device Under Test. During execution, registers will be collected from the real processor.
2. The collected register data will be compared with the corresponding outputs from the emulated fuzzing step (or other-word corpus generation). Differences in coverage could indicate unexpected ARM behavior, potentially exposing bugs or vulnerabilities. These cases require manual testing.

4.3. Creation of Test Input

A critical component of hardware fuzzing is generating valid and diverse initial test cases — commonly referred to as the seed corpus. In the context of HARMFuzz, these test cases are ELF binaries intended for bare-metal ARM execution. As discussed in Section 2.3, ELF files have a strict and well-defined structure. Any improper modification — such as altering headers or increasing the file size — may corrupt the binary and cause it to malfunction. Since these binaries are used as bare-metal applications, proper initialization during the boot process is essential. If the binary fails to boot correctly, the input may not execute at all, resulting in zero coverage and rendering the fuzzing process ineffective. Moreover, such failures can mislead HARMFuzz into incorrectly attributing the malfunction to a processor bug, when in fact it was caused by a corrupted ELF file.

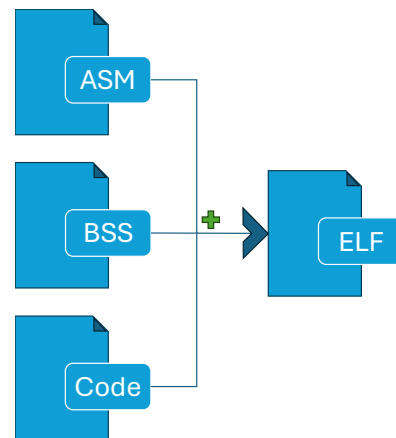


Figure 4.2.: Creation of the first input files

It is important to minimize the generation of corrupted test cases, because HARMFuzz’s mutation algorithm relies on existing files to produce new ELF binaries. If a corrupted file is used as input, it will likely generate more corrupted binaries, which will also be stored. This creates a chain reaction that reduces HARMFuzz’s effectiveness by wasting resources

on invalid or uninteresting test cases, making the detection of genuine anomalies more difficult and time-consuming.

Furthermore, the diversity of the initial corpus significantly impacts the fuzzer's ability to quickly reach unexplored code paths. A strong and diverse seed corpus enables more effective mutations, accelerates coverage growth, and helps avoid early stagnation — a state in which the mutator generates mostly uninteresting test cases that fail to discover new paths or improve coverage.

To ensure the quality and usefulness of the test inputs, the ELF File must be generated as shown in Figure 4.2; it should be generated from 3 initial files.[2.3]

1. **Initialization Assembly File.** This low-level startup code performs hardware-specific initialization, sets the stack pointer, configures memory-mapped peripherals, and prepares the environment to run higher-level C code. Since each ARM device may differ in its memory layout or peripheral base addresses, this file must be written carefully and remain unchanged during fuzzing.
2. **The Linker Script** defines how various sections of the ELF file (e.g., `.text`, `.data`, `.bss`) are mapped into memory. It specifies load addresses and aligns memory segments so that the bootloader or startup code can correctly interpret the layout. This script is essential to ensure that the application runs properly within the device's memory map (refer to §2.3 for more details).
3. **Main Code File (Payload).** This contains the C code that will be executed on the device after boot. Only this section is subject to mutation. During fuzzing, the mutation engine will generate variations of this part to explore different program behaviors and code paths. Further this file will be named as **main** section of ELF File.

These three files are compiled into a single ELF binary, which is then used both in the emulated processor and on the real processor.

The first two files are responsible for setting up the system's memory and peripherals. Because of this, these parts must remain unchanged during mutation to ensure proper functionality.

As discussed in the ELF File Structure (§ 2.3), the ELF format is complex, with each section depending on the previous ones. Any modification to these dependencies could lead to unintended behavior or corruption of the file structure.

```

1 000000000080000 <_start>:
2 000000000080050 <main>:
3     80050: a9bf7bfd      stp     x29, x30, [sp, #-16]!
4     80054: 910003fd      mov     x29, sp
5     80058: 94000008      bl      80078 <uart_init>
6     8005c: 90000000      adrp    x0, 80000 <_start>
7     80060: 91088000      add     x0, x0, #0x220
8     80064: 94000057      bl      801c0 <uart_puts>
9     80068: 94000048      bl      80188 <uart_getc>
10    8006c: 12001c00      and     w0, w0, #0xff
11    80070: 9400003c      bl      80160 <uart_send>
12    80074: 17ffffff      b       80068 <main+0x18>
13 000000000080078 <uart_init>:
14 000000000080160 <uart_send>:
15 000000000080188 <uart_getc>:
16 0000000000801c0 <uart_puts>:

```

Figure 4.3.: Test file example

As shown in Figure 4.3, the ELF contains multiple sections. The `.text` section includes six parts: the `_start` is the entry point of a bare-metal application and marks the beginning of the system bootstrap code; `uart_*` sections are peripherals. This section must remain the same after each mutation; the changing part is only the `main` section. However, after mutation, this section must not be bigger or smaller than the defined size of the section from the program header because it will overwrite memory sections that are given to other sections, which causes memory allocation problems due to improper interpretation of the ELF file.

4.4. Coverage Collection

HARMFuzz uses a two-stage coverage collection process using QEMU and QEMU-KVM to optimize fuzzing performance.

In the first stage, QEMU is used to emulate ARM hardware. This allows for fast test case generation on a high-performance machine, bypassing the complexity of hardware coverage collection. Code coverage is easily collected via `AFL++` and `afl-clang-fast` instrumentation, enabling efficient execution path tracking with minimal performance

overhead. Emulation allows faster and more flexible testing compared to the slower, more complex hardware coverage collection.

In the second stage, HARMFuzz tests on a real ARM CPU (Raspberry Pi 4B) using QEMU-KVM. KVM provides access to real CPU registers, which is crucial for monitoring processor behavior. Together, QEMU and KVM enable Register State Tracing, offering information on data flow without the need for additional hardware complexity. During this stage, a comparison between the register states of the emulated and real hardware is happened, helping to identify potential bugs or vulnerabilities.

Overall, QEMU speeds up fuzzing by allowing register tracing in an emulated environment. It doesn't directly collect code coverage, but it allows tools such as AFL++ to collect coverage on emulated hardware. Additionally, QEMU-KVM adds the benefit of testing on real CPU with register tracing, giving insight into both software and hardware behavior.

4.5. Design Considerations

The primary goal of HARMFuzz is to develop an efficient ARM-focused fuzzer capable of detecting hardware-specific bugs and vulnerabilities by combining real hardware interaction with emulation. However, HARMFuzz faces several hardware-specific design challenges. Below we outline each problem and the conceptual solution proposed in this thesis.

1. **Mutation Strategy** – Traditional fuzzing mutation engines, such as those used in AFL++, are designed for general-purpose binaries and often perform random or offset-based byte-level mutations without awareness of the internal structure of ELF files. When such generic mutation strategies are applied to ELF binaries—especially for bare-metal execution—they frequently produce corrupted files.

This is particularly problematic in HARMFuzz, where the test cases are ELF files, and must be proper structured. For example, if a generic mutator blindly inserts bytes into the `.text` or `main` section to create variation, it may shift subsequent sections like `.bss` or `.data` without adjusting offsets or headers accordingly. This corruption leads to faulty memory layout, which prevents the binary from booting correctly. In practice, this results in zero code coverage and a false signal to the fuzzer that the input triggered a bug, when in fact it just failed to execute.

HARMFuzz addresses this challenge with a structure-aware mutation strategy. It uses the full range of combinational mutation techniques provided by AFL++, but restricts their application strictly to the `main` section of the ELF file. Crucially, HARMFuzz ensures that the overall size of the ELF file remains unchanged, no shrinking or expansion occurs during mutation. This is achieved through a carefully designed format of initial test cases, which allows the system to dynamically identify the boundaries of the `main` section at runtime. By performing all mutations to this segment and preserving structure of headers and memory sections, HARMFuzz maintains the structural integrity of the binary while still enabling meaningful input variation.

2. **Hardware Emulation** - Hardware ARM emulation must be done with proper instruments that were carefully designed by developers so they can be used as an emulated model for input generation and comparison of the chip's behavior. It must be reliable with minimal inaccuracies.

HARMFuzz solves this challenge by QEMU usage, a mature, actively maintained, and open-source system emulator widely used in research and industry. To improve emulation accuracy and reduce noise from irrelevant peripherals, HARMFuzz applies a minimal hardware configuration to QEMU. This configuration excludes non-essential device models and peripheral components, focusing solely on the core CPU architecture and memory behavior relevant to the test environment. With this customized configuration, the differences between the emulated and real executions are reduced, and deviations are more likely to be caused by real hardware anomalies rather than emulator problems.

3. **Hardware Coverage Collection** - One significant challenge in hardware fuzzing is defining and collecting coverage metrics. Coverage must be captured throughout the entire kernel execution cycle, from boot process initialization to device shutdown. Collecting both register states and code coverage presents difficulties because monitoring register states in a real processor is not straightforward, particularly during the pre-boot phase.

HARMFuzz solves this challenge by combination of code coverage and emulated hardware execution. In standard QEMU emulation, coverage collection is relatively straightforward due to the virtualized environment. However, HARMFuzz extends this capability to native behavior of processor by using QEMU with KVM (Kernel-based Virtual Machine).[2.4] This configuration allows the system to use the actual physical processor for instruction execution while keeping peripheral hardware

emulated. This hybrid approach makes it possible to collect detailed code coverage even when using a real processor.

Additionally, both QEMU and QEMU KVM provide register tracing capabilities, including the ability to record register states after each instruction execution. This gives valuable runtime information throughout the entire execution flow, including during early boot phases that are typically difficult to instrument on physical hardware. By capturing both control flow and register state changes from startup to shutdown, HARMFuzz builds a more complete picture of system behavior during fuzzing operations, and can compare behaviour between emulated processor from QEMU and real processor from KVM.

4. **Fuzzing Platform** - Developing a fuzzer from scratch is a difficult and time-consuming process, especially if the fuzzer is competitive and efficient. Because of this, the initial platform for the fuzzer must be scalable and versatile.

HARMFuzz integrates AFL++ as a backend for mutation and corpus management. The emulator (QEMU) is augmented to support register tracing and instruction-level logging. The modularity of the design allows integration with additional peripherals or future enhancements.

5. Implementations

This section provides detailed implementations of the various components that constitute HARMFuzz. Each component plays a specific role in the framework, contributing to its overall functionality and performance. The implementation details include architectural design decisions, key algorithms, data structures, and the interactions between components.

5.1. Software

After careful testing and comparison, the initial software and instrumentation setup for HARMFuzz is as follows:

1. **Fuzzing Platform** - The initial platform for fuzzing development is AFL++ [54], a large and popular open-source fuzzer maintained by the GitHub community. AFL++ is based on the standard AFL but offers more options for coverage tools and mutation techniques. This makes AFL++ ideal for HARMFuzz as it is fully customizable and also provides a variety of instruments for coverage collection.
2. **Hardware Simulation and Hardware Coverage Collection** – QEMU is used for ARM hardware emulation, primarily because it supports KVM, which enables coverage data collection directly from the CPU starting at the earliest stages of the boot process. Furthermore, QEMU is an open-source platform that is actively maintained and regularly updated by a large community of developers, enhancing the reliability of the emulation. The use of fully emulated hardware also makes it possible to rely on code coverage instead of complex register-level coverage. This approach increases the speed of software fuzzing while avoiding the overhead typically associated with hardware-level fuzzing.

-
3. **Hardware Simulation and Hardware Coverage Collection** – QEMU is used for ARM hardware emulation, primarily because it supports KVM, which enables coverage data collection directly from the CPU starting at the earliest stages of the boot process. Furthermore, QEMU is an open-source platform that is actively maintained and regularly updated by a large community of developers, enhancing the reliability of the emulation. The use of fully emulated hardware also makes it possible to rely on code coverage instead of complex register-level coverage. This approach increases the speed of software fuzzing while avoiding the overhead typically associated with hardware-level fuzzing.

5.1.1. Mutation Implementation of HARMFuzz

The mutation engine in HARMFuzz is responsible for generating diverse and valid test inputs to stimulate different CPU execution paths. HARMFuzz builds upon the mutation strategy of AFL++, with several significant modifications adapted for ARM-based fuzzing, particularly in a structured binary (ELF) [2.3] context.

Standard AFL++ is optimized for fuzzing unstructured binary or plain text inputs. However, when used on ELF files, it introduces two major issues:

- **Trimming:** AFL++ [54] tries to reduce input size to speed up execution, but this behavior corrupts ELF file structure [2.3], as even slight changes in ELF headers or section alignments render the binary invalid. This was mitigated by setting the environment variable `AFL_DISABLE_TRIM=1` during fuzzing runs, which disables the trimming stage.
- **Unrestricted Mutation:** By default, AFL++ mutates the entire input file. When applied to structured files like ELF [2.3], this behavior destroys headers or code sections critical to successful execution in QEMU KVM. To solve this, HARMFuzz uses custom mutator, which mutates the program's `main` instruction section only.

Mutation Range Identification. Custom linker scripts, startup assembly files, and BSS definitions [4.2] ensure that the executable code to be fuzzed is always placed at a known location in the ELF binary (starting at `0x10030`). This controlled placement allows the mutator to reliably identify the beginning of the fuzzing region, which would not be possible otherwise.

To determine the endpoint dynamically, the ARM return instruction `0xC0035FD6` is searched within the ELF text section. This instruction indicates the end of the main

function. Due to a special pattern used for generating the program's `main`, C code payload, instruction section, it consistently ends with this return statement, enabling reliable detection of the `main` section's endpoint. The mutation range is identified using the logic described in Algorithm 1, which scans the ELF file from a fixed start address until the first return instruction is found.

Algorithm 1 Developed HARMFuzz Mutator

Require: $start_address \leftarrow 0x10030$

```

1:  $end\_address \leftarrow start\_address + 0x4$  ▷ The end address is the second instruction in
   the main section
2:  $i \leftarrow start\_address$ 
3: while  $i \leq$  address of last instruction of ELF file do
4:   if instruction[ $i$ ] is a return instruction then
5:      $end\_address \leftarrow i$ 
6:     STOP While loop
7:   end if
8:    $i \leftarrow i + 0x4$ 
9: end while
10:  $length\_of\_mutated\_code \leftarrow end\_address - start\_address$ 
11: AFL++ mutates the ELF file starting from  $start\_address$ , it should mutate only
    instructions in the range of  $length\_of\_mutated\_code$ , without changing or removing
    return instruction

```

In addition to the standard AFL++ mutations, HARMFuzz implements custom strategies designed to maintain the integrity of ARM machine code instructions. Standard AFL++ mutations can easily break the structure of instructions (such as bit-flip), leading to invalid program behavior or ELF file loading failures. To address this, HARMFuzz introduces instruction-aware mutations that ensure the executable code remains syntactically valid and operational.

The primary advanced mutation strategies include:

- **Insertion of NOP instructions:** NOP instructions, or their functional equivalents in ARM, are inserted at random positions within the valid mutation range. This allows for variation in code density and changes in cache behavior without altering the program's logic.
- **Instruction reordering:** Instructions within the main section are shuffled (with constraints), ensuring that the size and alignment are maintained, which enables

the exploration of alternative execution paths.

- **Operand mutation:** Registers or immediate values within existing instructions are substituted, keeping the type of instruction intact. This can include modifying the operands of arithmetic or logical operations without changing the core operation itself.
- **Insertion of randomly generated instructions:** Valid, syntactically correct, and safe instructions are randomly inserted at specific points, introducing new paths of execution and increasing fuzzing coverage.

These mutation strategies are implemented as a custom AFL++ mutator written in C, which, after determining the mutation range (see 1A), parses the ELF file, decodes the instructions, and applies safe, architecture-specific modifications. This approach ensures that the input data remains executable while increasing path diversity and coverage, without disrupting the ELF file structure.

This section presents only Algorithm 1 of HARMFuzz Mutator. More detailed code can be found in Appendix (§ A)

5.1.2. Coverage Collection

In HARMFuzz, Coverage Collection is a crucial part of the fuzzing process. The fuzzing happens in two stages: the first on emulated hardware in QEMU, and the second on real hardware (Raspberry Pi 4 Model B) in QEMU KVM.

The first stage is executed on QEMU [2.4], a tool that fully emulates a computer system, including peripherals. This enables fuzzing of hardware as software, bypassing the complexity of hardware coverage collection. During this phase, Code Coverage is used, coverage tracing is performed by AFL++ using afl-clang-fast instrumentation, the process of adding special code flags to the target program that enables execution paths tracking such as which parts of the code are executed and how often. Specifically, afl-clang-fast is efficient instrumentation without significant performance overhead.

QEMU makes it easy to monitor executable code, ensuring that different parts of the system are tested, and different execution paths are explored to identify potential errors.

In the second stage, HARMFuzz focuses on the interesting test cases discovered in Stage 1, which triggered higher coverage, crashes, or timeouts. These cases are run on QEMU

KVM on a Raspberry Pi 4B. The use of KVM is important because it uses a real CPU, while QEMU provides the flexibility to enable advanced tracing capabilities.

QEMU allows for tracking CPU registers after each instruction, offering valuable insights into the processor's behavior and the movement of data through the system. This capability is especially useful for monitoring the early stages of booting, a process that is difficult to observe on real hardware.

The register state tracing is facilitated by QEMU, GDB, and a custom shell script developed specifically for HARMFuzz. This shell script collects the CPU registers after each instruction is executed, enabling detailed monitoring of how the processor's state changes throughout the fuzzing process.

During this stage, both register coverage from QEMU KVM and the emulated CPU are tracked. This comparison of traces from the real CPU and the emulated environment aids in detecting differences, which may be caused by bugs or vulnerabilities.

5.1.3. QEMU and QEMU-KVM Configuration Run

QEMU serves as the primary tool for emulating and testing ARM-based kernels. In this project, QEMU and QEMU-KVM are utilized for differential hardware fuzzing on ARM systems, focusing on the Cortex-A72 Processor that is used in Device Under Test (Raspberry Pi 4 Model B)

The QEMU command for running the kernel is as follows:

```
qemu-system-aarch64 -machine virt -cpu cortex-a72 -display none -  
kernel @@
```

This configuration specifies the following:

- **-machine virt:** The `virt` machine type is an ARM-specific virtual platform that supports KVM acceleration. This feature is important for HARMFuzz, as it allows the use of a real ARM CPU for better accuracy, while still providing the flexibility of emulation and advanced tracing capabilities.
- **-cpu cortex-a72:** This explicitly defines the emulated CPU as Cortex-A72, that is used in Device Under Test (Raspberry Pi 4 Model B). As a high-performance ARM core, the Cortex-A72 is widely used in modern ARM-based systems, making it an ideal target for differential fuzzing experiments.

-
- **-display none**: Disables the graphical display, as it is not needed for kernel execution.
 - **-kernel @@**: The placeholder @@ represents the kernel file to be loaded for execution. In practice, during HARMFuzz, @@ will be used as a specified placeholder, indicating where the fuzzer will insert its test cases. This allows the fuzzer to dynamically place the generated test cases into the appropriate location within the kernel image for testing. If we were using QEMU directly without HARMFuzz, this placeholder would be replaced with the path to the compiled kernel image being tested (e.g., `-kernel /path/to/kernel.img`).

For hardware-accelerated runs using KVM (Kernel-based Virtual Machine), the configuration is as follows:

```
1 qemu-system-aarch64 -machine virt,accel=kvm -cpu host -display none -kernel
```

Here, the key difference is:

-machine virt,accel=kvm -cpu host: The virt machine type is combined with KVM acceleration. KVM allows QEMU to run the guest kernel directly on the host's hardware, which allows the collection of coverage information directly from a real ARM CPU. The virtual machine type is critical here because it fully supports KVM, ensuring optimal hardware interaction.

5.2. Hardware

Implementation of HARMFuzz for hardware fuzzing is not a complex process, but it must meet several criteria. First of all, the Device Under Test (DUT) must feature an ARM processor, as HARMFuzz is specifically designed to work with ARM architectures. Secondly, the device must support KVM virtualization, which is essential for using the real host ARM CPU instead of relying on a fully virtualized one. This allows for more accurate fuzzing results by minimizing differences between the emulated and real hardware environments. Additionally, the CPU should be a popular and widely adopted ARM core. This ensures the relevance of the fuzzing results and facilitates broader applicability. The more accessible the platform, the easier it will be to reproduce research.

5.2.1. Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B is an excellent choice for hardware fuzzing with HARMFuzz due to its compliance with all given criteria. It is equipped with a Broadcom BCM2711

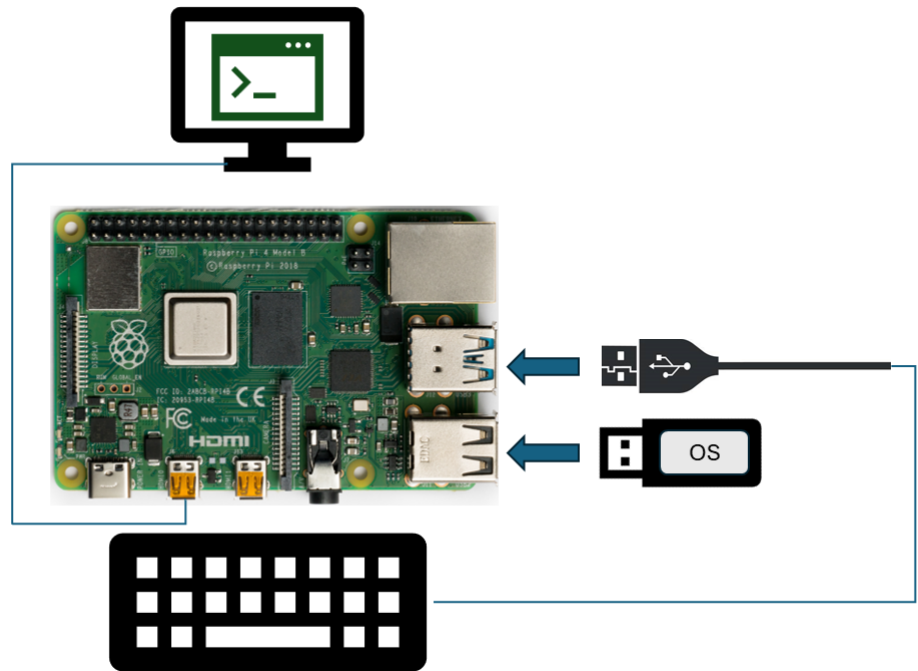


Figure 5.1.: Raspberry Pi 4 Model B Setup

System-on-Chip (SoC), which integrates a Cortex-A72 quad-core ARM processor. As HARMFuzz is specifically designed for ARM architectures, the Cortex-A72 core makes the Raspberry Pi 4 a perfect fit. The Raspberry Pi 4 Model B has KVM (Kernel-based Virtual Machine) support, HARMFuzz can directly access the host's physical ARM processor, ensuring that fuzzing results reflect the actual hardware behavior that could be used for comparison with emulated Cortex-A72. The CPU in the Raspberry Pi 4 is a modern and widely used ARM core. It balances high performance and energy efficiency, making it a popular choice for both developers and researchers. The Raspberry Pi platform's affordability and widespread adoption further enhance its appeal, allowing researchers to

easily replicate experiments and extend findings to other ARM devices.

The end connection is as shown in Figure 5.1; it does not include the external power supply, which powers the Raspberry Pi 4 Model B during the fuzzing process.

Overall, the hardware setup is simple: a USB flash drive with the Raspberry Pi operating system. Any OS could be used, but this project specifically uses Raspberry Pi OS Lite (64-bit), which does not include a desktop environment—something unnecessary for HARMFuzz. Additionally, the only I/O devices used are a keyboard and a monitor.

That shows that initial equipment for HARMFuzz is minimal and doesn't need any special equipment that could not be found at home, except of Device Under Test. Fortunately, many of modern equipment uses ARM cores, so even DUT could be easily found at home, and be used if it has KVM Virtualization Support.

6. Performance Evaluation

This section compares HARMFuzz with the standard AFL++ Fuzzer in two high-performance environments: one using only QEMU and the other utilizing QEMU with KVM on a Raspberry Pi 4 Model B. Both QEMU setups were instrumented with **afl-clang-fast** and **afl-clang-fast++** to collect coverage data. This instrumentation method is a standard choice for AFL++ and is also employed for coverage collection in HARMFuzz.

The primary metric of comparison is the number of edges discovered over time, which indicates the amount of code triggered during execution—commonly referred to as code coverage. Additionally, this evaluation includes a comparison between code coverage and register coverage to demonstrate that code coverage is a reliable metric for processor evaluation. The execution speeds of both fuzzers under both conditions are also analyzed.

The data for this evaluation was collected over a period of four days to ensure clarity and reliability. A set of 82 bare-metal applications, specifically designed for a virtual machine environment, served as input files. The large corpus ensures the proper operation of both fuzzers.

Although HARMFuzz is based on AFL++, it was designed and optimized specifically for ARM fuzzing. AFL++ is a well-known fuzzing tool that works well for general fuzzing tasks, making it a good baseline for comparison. The main difference between them is that HARMFuzz has special optimizations for ARM architecture, allowing it to target ARM-specific code coverage more effectively. By comparing HARMFuzz to AFL++, we show its strengths in improving fuzzing coverage and performance on ARM systems, while also recognizing the performance trade-offs from preserving input file integrity.

At the end we generated around 4000 new test cases in QEMU and around 1000 in QEMU KVM.

As shown in Figures 6.1 and 6.2, HARMFuzz consistently outperforms AFL++ in terms of total code coverage. Specifically, HARMFuzz identifies approximately 1,000 additional edges in the QEMU environment and around 100 additional edges when running in the

QEMU with KVM. These results demonstrate that HARMFuzz is more effective in exploring the target program's execution paths, making it a valuable tool for fuzzing ARM-based software where coverage is a critical metric.

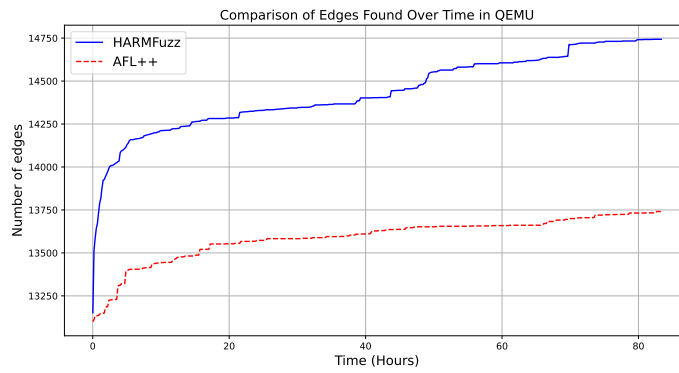


Figure 6.1.: Comparison coverage in QEMU

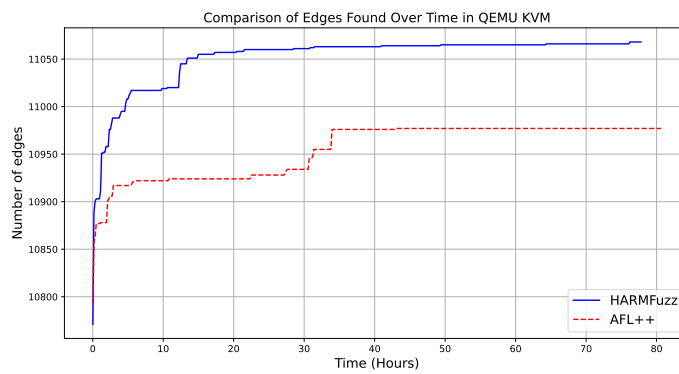


Figure 6.2.: Comparison coverage in QEMU KVM

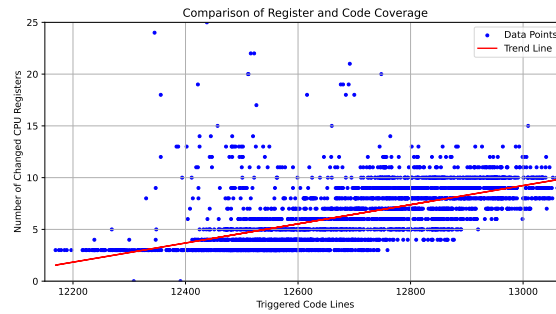


Figure 6.3.: Comparison of Register and Code Coverage

In Figure 6.3, we show the relationship between code coverage and register coverage. The results clearly indicate that register coverage increases as code coverage rises, following an overall upward trend. This suggests that code coverage is a good metric for evaluating processor performance and can be used hardware coverage metric. To further support

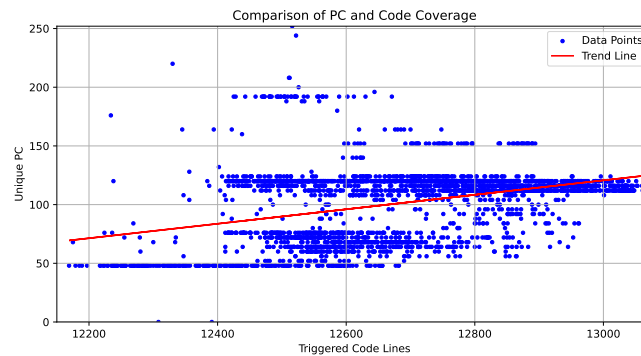


Figure 6.4.: Comparison of PC and Code Coverage

this, we compared code coverage with the number of unique program counters triggered during corpus execution. This comparison helps prove that code coverage is not only useful for measuring general execution, but also for capturing variations in instruction flow.

As shown in Figure 6.4, the trend continues to rise, confirming that code coverage is a reliable metric for processor evaluation. This proves that code coverage is a valuable and

reliable tool in assessing the effectiveness of hardware fuzzing, as it provides a clear metric for exploring and testing various hardware configurations and responses.

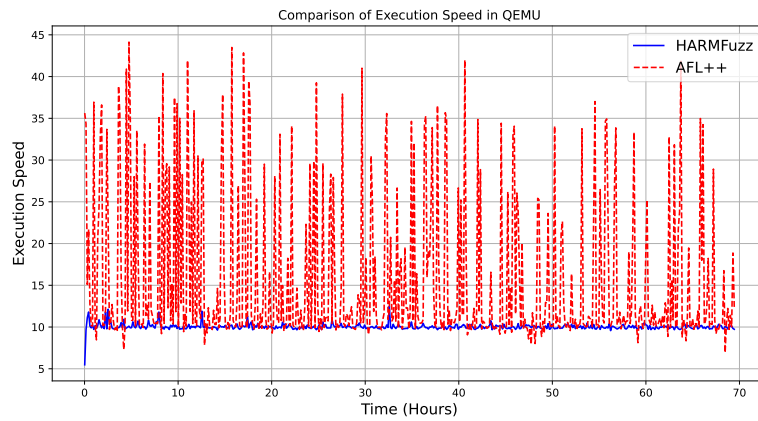


Figure 6.5.: Comparison speed in QEMU

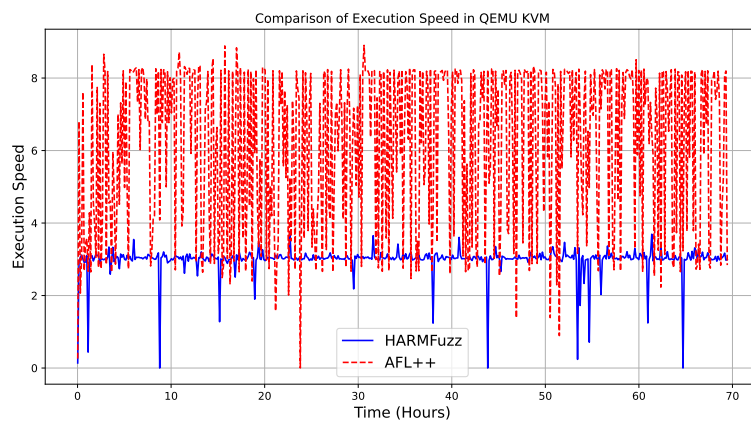



Figure 6.6.: Comparison speed in QEMU KVM

Figures 6.5 and 6.6 highlight the execution speed of both fuzzers. Average speed of HARMFuzz is 10 executions per second against 22 executions per second in AFL++ in QEMU, and 3 executions per second against 5 in QEMU KVM. HARMFuzz is slower than



AFL++ due to the absence of the trimming process, which is omitted in HARMFuzz to preserve the structure of ELF files. While this increases execution time, it is a necessary trade-off to ensure the correctness and usability of input files as kernels during QEMU or QEMU KVM runs.

7. Discussion

HARMFuzz’s design is rooted in the principles of differential fuzzing, using a combination of emulated ARM hardware environment and real device to detect hardware-specific bugs and vulnerabilities. The workflow begins with the generation of ELF files, carefully structured to maintain compatibility with both simulation and real hardware. These files are tested on emulated environments using QEMU without KVM and real hardware using QEMU with KVM, enabling a comparison of coverage results and register states. Feedback from these tests guides the mutation process, ensuring that the input files are continuously refined to trigger new behaviors and uncover potential vulnerabilities.

7.1. Advantages

HARMFuzz offers several key advantages that demonstrate its effectiveness and novelty. First of all, **Enhanced Code Coverage**, the differential fuzzing approach in HARMFuzz leads to consistently higher code coverage compared to standard AFL++, identifying 1,000 additional edges in QEMU environments and 100 in KVM-enabled setups. This improved coverage ensures a more thorough exploration of code paths. Secondly, **focus for ARM Architectures**, by focusing on ARM-specific challenges, HARMFuzz maintains the structural integrity of ELF files during mutation, ensuring compatibility with ARM processors. This design surpasses generic fuzzing tools, which often struggle with structured inputs. Moreover, using QEMU with KVM, HARMFuzz achieves efficient register data collection, particularly during critical phases like the boot process. This capability allows it to detect hardware-specific bugs that might be unnoticed in purely emulated setups. Additionally, HARMFuzz is designed to work with widely available and affordable hardware, such as the Raspberry Pi 4 Model B. This accessibility makes it easier for researchers and developers to replicate experiments and extend the findings to other ARM-based systems.

7.2. Limitations

While HARMFuzz solves many challenges in ARM fuzzing, it is not without limitations. A key limitation of HARMFuzz lies in its reliance on hardware emulation and virtualization support. Without these, HARMFuzz cannot be used for testing ARM processors. This dependency restricts its applicability to environments where such features are available, presenting a significant obstacle to broader adoption.

An additional problem is the speed of HARMFuzz. The absence of a trimming mechanism in HARMFuzz, while necessary to maintain the structural integrity of ELF files, results in slower execution speeds compared to standard fuzzers like AFL++. This slower speed may become a bottleneck in scenarios requiring rapid feedback or extensive testing of large-scale applications.

8. Conclusion

HARMFuzz represents a significant advancement in ARM-specific fuzzing by addressing key limitations of standard AFL++ in handling structured input files like ELF. Through its scoped mutation strategy tailored for ELF files, HARMFuzz maintains the integrity and proper execution of inputs, ensuring compatibility with execution environments such as QEMU and QEMU KVM. This approach enables more effective fuzzing of ARM processors by generating syntactically valid and semantically meaningful inputs, which in turn facilitates higher coverage and deeper program exploration.

A key innovation in HARMFuzz is its use of emulation to collect high-quality test cases, thereby avoiding the complexity of hardware coverage collection. By relying on a direct proxy of CPU complexity in the software domain, it enables efficient tracing and analysis of CPU behavior without the need for additional physical equipment. This method allows for more efficient fuzzing runs, focusing resources on valuable execution paths rather than low-level hardware interactions.

However, this efficiency comes with some trade-offs. The absence of a trimming process—necessary to preserve the structured nature of ELF inputs—results in reduced execution speed. Additionally, HARMFuzz’s reliance on hardware emulation and virtualization may limit its applicability in broader testing scenarios. Despite these challenges, HARMFuzz has demonstrated superior performance by achieving higher coverage within the same time period, highlighting its effectiveness in precise and reliability-critical contexts.

Future work can focus on addressing current limitations by optimizing execution speed, extending support to other processor architectures, and reducing dependence on hardware-specific features. These improvements could further enhance HARMFuzz’s potential as a versatile and scalable solution for automated vulnerability detection in ARM-based systems.

8.1. Acknowledgment

During the writing of this bachelor's thesis, large language model (LLM) tools were used exclusively for rephrasing and grammar correction to enhance the lexical and stylistic quality of the text. They were not used to generate original content or contribute to the research itself.

A. Appendix Developed Mutator

```
1  const size_t start_offset = 0x10030;
2
3  size_t afl_custom_fuzz(my_mutator_t *data, uint8_t *buf, size_t
4  buf_size, u8 **out_buf, uint8_t *add_buf, size_t add_buf_size,
5  size_t max_size) {
6      if (max_size > data->buf_size) {
7          u8 *ptr = realloc(data->buf, max_size);
8          if (!ptr) {
9              return 0;
10         } else {
11             data->buf = ptr;
12             data->buf_size = max_size;
13         }
14     }
15     u32 havoc_steps = 1 + rand_below(data->afl, 16);
16     memcpy(data->buf, buf, buf_size);
17     size_t end_offset = start_offset+0x4;;
18     size_t i = start_offset;
19     while(i+0x3 < buf_size){
20         if (data->buf[i+2] == 0x5f && data->buf[i+3] == 0xd6 &&
21             data->buf[i] == 0xc0 && data->buf[i+1] == 0x03) {
22             end_offset = i;
23             break;
24         }
25         i=i+0x4;
26     }
27     size_t segment_length= end_offset - start_offset;
28     u32 mutated_segment_len = afl_mutate(data->afl, data->buf +
29     start_offset, segment_length, havoc_steps, false, true,
30     add_buf, add_buf_size, max_size);
31     *out_buf = data->buf;
32     return buf_size;
33 }
```

Bibliography

- [1] Ned Batchelder. *Coverage.py*. <https://coverage.readthedocs.io/en/7.6.10/>. [Online; accessed 19-January-2025]. 2025.
- [2] Bijit Ghosh. *Fuzz Testing*. <https://medium.com/@bijit211987/fuzz-testing-d47d1de47f98>. [Online; accessed 04-05-2025]. 2022.
- [3] Pallavi Borkar et al. “WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5377–5394. ISBN: 978-1-939133-44-1. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/borkar>.
- [4] Sadullah Canakci et al. “ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance”. In: *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2023, pp. 1–12. DOI: 10.1109/HOST55118.2023.10133714.
- [5] Clang contributors. *AddressSanitizer*. [Online; accessed 13-May-2025]. URL: <https://clang.llvm.org/docs/AddressSanitizer.html>.
- [6] Clang contributors. *LeakSanitizer*. [Online; accessed 13-May-2025]. URL: <https://clang.llvm.org/docs/LeakSanitizer.html>.
- [7] Clang contributors. *ThreadSanitizer*. [Online; accessed 13-May-2025]. URL: <https://clang.llvm.org/docs/ThreadSanitizer.html>.
- [8] cloudflare. *What is Meltdown/Spectre?* <https://www.cloudflare.com/learning/security/threats/meltdown-spectre/>. [Online; accessed 25-February-2025].
- [9] Zachary Comeau. *This Hacking Story Illuminates Why Cybersecurity is So Important in Smart Homes*. <https://www.cepro.com/news/this-hacking-story-illuminates-why-cybersecurity-is-so-important-in-smart-homes/>. [Online; accessed 4-February-2025]. 2024.

-
-
- [10] Christopher Domas. “Breaking the x86 ISA”. In: *Black Hat 1* (2017), pp. 1–6.
- [11] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. *FormatFuzzer: Effective Fuzzing of Binary File Formats*. 2023. arXiv: 2109.11277 [cs.SE]. URL: <https://arxiv.org/abs/2109.11277>.
- [12] Andrea Fioraldi and Dominik Maier. *The LibAFL Fuzzing Library*. [Online; accessed 4-May-2025]. URL: https://aflplus.plus/libafl-book/core_concepts/mutator.html.
- [13] Free Software Foundation. *gcov—a Test Coverage Program*. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>. [Online; accessed 19-January-2025]. 2025.
- [14] Jian Gao et al. “Em-fuzz: Augmented firmware fuzzing via memory checking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3420–3432.
- [15] Moritz Lipp; Michael Schwarz; Daniel Gruss; Thomas Prescher; Werner Haas; Anders Fogh; Jann Horn; Stefan Mangard; Paul Kocher; Daniel Genkin; Yuval Yarom; Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.
- [16] Kyle Hanslovan. *Fuzzing for Quality Assurance*. <https://kylehanslovan.medium.com/fuzzing-for-quality-software-12212cd75fb4>. [Online; accessed 04-05-2025]. 2016.
- [17] Joran Honig. *Fuzzing - White/Black and Grey Boxes*. <https://typeshare.co/joranhonig/posts/fuzzing---101---boxes>. [Online; accessed 04-05-2025]. 2022.
- [18] Rahul Kande et al. “{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3219–3236.
- [19] Koffi Anderson Koffi et al. “StructuredFuzzer: Fuzzing Structured Text-Based Control Logic Applications”. In: *Electronics* 13.13 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13132475. URL: <https://www.mdpi.com/2079-9292/13/13/2475>.
- [20] Martin Kot. *The LibAFL Fuzzing Library*. 2003.
- [21] Doug Kwan et al. “SiliFuzz: fuzzing CPUs by proxy”. In: *Google Research* (2021).
- [22] Xixing Li et al. “Uisfuzz: An efficient fuzzing method for cpu undocumented instruction searching”. In: *IEEE Access* 7 (2019), pp. 149224–149236.

-
-
- [23] Shan Lu et al. “BugBench: Benchmarks for Evaluating Bug Detection Tools”. In: (Jan. 2005).
- [24] Andrea Fioraldi Marc van Hauser Heuse Dominik Maier. *AFL++*. <https://github.com/AFLplusplus/AFLplusplus>. [Online; accessed 4-February-2025]. 2025.
- [25] Natalie Muradyan. “PCBleed: Fuzzing for CPU Bugs Through Use of Performance Counters”. PhD thesis. Massachusetts Institute of Technology, 2024.
- [26] Tavis Ormandy. *Das Bild der TU Darmstadt*. 2023. URL: <https://lock.cmpxchg8b.com/zenbleed.html>. (Accessed: 16.10.2024).
- [27] Maxime Peterlin. *Hyperpom: An Apple Silicon Fuzzer for 64-bit ARM Binaries*. 2022. URL: https://blog.impalabs.com/2211_hyperpom.html. (Accessed: 15.10.2024).
- [28] QEMU. *QEMU*. <https://www.qemu.org/>. [Online; accessed 4-February-2025]. 2025.
- [29] N.GUYEN Anh Quyn and DANG Hoang Vu. *Unicorn The Ultimate CPU emulator*. 2025. URL: <https://www.unicorn-engine.org/>.
- [30] Mohamadreza Rostami et al. “Fuzzerfly Effect: Hardware Fuzzing for Memory Safety”. In: *IEEE Security & Privacy* (2024).
- [31] Stephan van Schaik et al. “RIDL: Rogue In-Flight Data Load”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.
- [32] Michael Schwarz et al. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *CCS*. 2019.
- [33] Anton Shilov. *842 Chips Per Second: 6.7 Billion Arm-Based Chips Produced in Q4 2020*. <https://www.tomshardware.com/news/arm-6-7-billion-chips-per-quarter>. [Online; accessed 4-February-2025]. 2021.
- [34] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. “Cascade: CPU fuzzing via intricate program generation”. In: *Proc. 33rd USENIX Secur. Symp.* 2024, pp. 1–18.
- [35] Prashast Srivastava and Mathias Payer. “Gramatron: Effective grammar-aware fuzzing”. In: *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 2021, pp. 244–256.
- [36] Fredrik Strupe. *ARMshaker*. <https://github.com/frestr/armshaker>. [Online; accessed 4-February-2025]. 2022.
- [37] The Clang Team. *AddressSanitizer*. <https://clang.llvm.org/docs/AddressSanitizer.html>. [Online; accessed 19-January-2025]. 2025.

-
-
- [38] Fabian Thomas et al. *RISCVuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing*. 2024.
- [39] Kevin Townsend. *Vulnerability Allows Hackers to Unlock Smart Home Door Locks*. <https://www.securityweek.com/vulnerability-allows-hackers-unlock-smart-home-door-locks/>. [Online; accessed 4-February-2025]. 2019.
- [40] Timothy Trippel et al. “Fuzzing hardware like software”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3237–3254.
- [41] Mingzhe Wang et al. “Data Coverage for Guided Fuzzing”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2511–2526. ISBN: 978-1-939133-44-1. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/wang-mingzhe>.
- [42] Wikipedia contributors. *ARC (processor)* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 13-May-2025]. 2025. URL: [https://en.wikipedia.org/w/index.php?title=ARC_\(processor\)&oldid=1287103745](https://en.wikipedia.org/w/index.php?title=ARC_(processor)&oldid=1287103745).
- [43] Wikipedia contributors. *ARM architecture family* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=ARM_architecture_family&oldid=1273224276. [Online; accessed 4-February-2025]. 2025.
- [44] Wikipedia contributors. *Basic block* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 4-May-2025]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Basic_block&oldid=1238067209.
- [45] Wikipedia contributors. *Executable and Linkable Format* — *Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/w/index.php?title=Executable_and_Linkable_Format&oldid=1251383864. (Accessed: 30.10.2024).
- [46] Wikipedia contributors. *Foreshadow* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Foreshadow&oldid=1258392160>. [Online; accessed 24-March-2025]. 2024.
- [47] Wikipedia contributors. *Formal verification* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 4-May-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Formal_verification&oldid=1285719773.
- [48] Wikipedia contributors. *Fuzzing* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 5-November-2024]. 2024. URL: <https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=1249540069>.

-
-
- [49] Wikipedia contributors. *Fuzzing* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=1270167521>. [Online; accessed 27-March-2025]. 2025.
- [50] Wikipedia contributors. *IBM Power microprocessors* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 13-May-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=IBM_Power_microprocessors&oldid=1280191557.
- [51] Wikipedia contributors. *Manual testing* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 4-May-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Manual_testing&oldid=1272008985.
- [52] Wikipedia contributors. *Moore's law* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=1281650652. [Online; accessed 24-March-2025]. 2025.
- [53] Paul Kocher; Jann Horn; Anders Fogh; Daniel Genkin; Daniel Gruss; Werner Haas; Mike Hamburg; Moritz Lipp; Stefan Mangard; Thomas Prescher; Michael Schwarz; Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.
- [54] Michal Zalewski. *AFL*. <https://github.com/google/AFL>. [Online; accessed 18-January-2025]. 2013.
- [55] Andreas Zeller. *The Fuzzing Book*. [Online; accessed 4-May-2025]. 2025. URL: <https://www.fuzzingbook.org/beta/html/MutationFuzzer.html>.