École Polytechnique Fédérale de Lausanne

Dynamic Analysis for Embedded RISC-V Firmware

by Maximilian Mosler

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

External Expert

Florian Hofhammer
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 12, 2023

Acknowledgments

First and foremost I would like to thank my supervisor Florian Hofhammer for his continued support throughout the project. The insights he provided and the manner in which he did were thoughtful and invaluable. It was a pleasure to be able to learn from someone who clearly has a very sophisticated understanding of the field, but remained grounded positive in his expectations. Admittedly, this semester was one of the most challenging I have yet experienced mentally, which is likely also reflected in the subpar quality of this paper. However I am thankful for the motivation and support Florian provided.

Lausanne, June 12, 2023 Maximilian Mosler

Abstract

This paper presents a comprehensive survey of three significant advancements in the field of firmware fuzzing: Fuzzware, EmbedFuzz, and P2IM. Fuzzwareintroduces a novel framework for fuzzing processor-peripheral interfaces in microcontroller unit (MCU) architectures, specifically focusing on ARM Cortex-M architecture. EmbedFuzz extends the capabilities of Fuzzwareby incorporating timing-based interrupt modeling and customizable interrupt firing strategies. P2IM provides an abstract model for capturing access patterns and handling strategies for different types of peripheral registers in MCU architectures.

Building upon the existing contributions, this paper proposes an extension to EmbedFuzz that adds partial support for RISC-V architecture. The extended EmbedFuzz with partial RISC-V support enables firmware execution and testing in RISC-V-based systems, further enhancing the efficacy of firmware fuzzing techniques. Experimental results demonstrate the effectiveness and practicality of the proposed extension, establishing it as a valuable tool for security analysis and vulnerability detection in RISC-V embedded systems.

Contents

Chapter 1

Introduction

Embedded systems have permeated various industries, including transportation, healthcare, communication, and vital infrastructure, becoming an essential component of contemporary society. These systems encompass a diverse range of devices, from smart appliances to industrial control systems, operating autonomously and carrying out critical tasks that impact society as a whole. As a result, thorough testing of embedded devices for software bugs that may lead to security vulnerabilities remains crucial. Failure to mitigate these issues could pose significant risks to financial, public, and national security. For this sake, Fuzzware [10], P2IM [7], and EmbedFuzz [8], amongst others, have introduced a fuzzing suites capable of running and fuzzing micro-controller unit (MCU) firmware.

Moreover, due to substantial differences in MCUs across architectures, MCU fuzzing frameworks face an ever-increasing number of challenges in supporting both new and old architectures. Currently, the focus is on supporting the ARM architecture, which has relatively strict implementation specifications, making it easier to accommodate a wide variety of chips utilizing this architecture. However, as we expand EmbedFuzz's support to include other architectures, it becomes evident that supporting different implementations of the same architecture, as well as new architectures, can be a burdensome endeavor. To elaborate further, the main challenges we will discuss include memory maps, toolchain support, documentation availability, firmware availability, and Linux support.

To address these challenges, this report initially surveys various fuzzing frameworks and their innovative ideas, with a particular emphasis on extending architecture support. Additionally, as part of this project, the implementation of one such framework, EmbedFuzz [8], was extended to run a minimum working example, providing practical insights into the matter. The primary objective is to highlight specific challenges encountered during the implementation of RISC-V while providing a survey of future challenges that may arise for other architectures.

In conclusion, this paper presents a comprehensive overview of the challenges faced by state-

of-the-art emulators, such as EmbedFuzz, when extending firmware support, while also providing initial RISC-V-64bit firmware support under EmbedFuzz.

Chapter 2

Background

2.1 Fuzzing and dynamic analysis

To understand the goal of fuzzing frameworks, we provide a brief explanation of fuzzing along the lines of [3].

In general dynamic analysis, analyses runtime behavior of the software under test. It involves instrumenting the target application to gather information about its execution, such as code coverage, memory access patterns, and system calls. This data provides insights into the software's internal state and helps identify potential vulnerabilities or suspicious behavior triggered by the inputs [3].

As a part of dynamic analysis, fuzzing is an automated software testing technique that aims to uncover vulnerabilities and bugs in programs by subjecting them to a large volume of unexpected and potentially malformed inputs [3].

Moreover, it involves generating and injecting random or mutated inputs into a target software application, monitoring its behavior, and identifying instances of crashes, hangs, or unexpected output. By exploring the vast input space, fuzzing can effectively identify unknown vulnerabilities, including memory corruption, logic flaws, and protocol errors [3].

Specifically, greybox fuzzing leverages limited knowledge of the target application's internal structure, such as API signatures, to guide the generation of more meaningful test inputs. By maintaining a balance between random inputs and guided exploration, greybox fuzzing significantly improves code coverage and increases the likelihood of discovering security-critical bugs [3].

2.2 Rehosting basics

We introduce the fundamental terminology and basics required to understand and clearly reason about the rehosting process. Moreover, we will do this by closely following the terminology and definitions used in "SoK: Enabling Security Analyses of Embedded Systems via Rehosting" [6].

The primary objective of rehosting is to establish a dynamic analysis process for embedded systems. This is achieved by devising a categorization model that enables the subsequent rehosting of these systems in software and/or hardware with a reasonable degree of accuracy. It is important to ensure a reasonable level of accuracy to ensure the meaningfulness of the dynamic analysis results. Poor accuracy may lead to over-approximation, resulting in the omission or inclusion of bugs, thus compromising the validity of the analysis outcomes [6].

To accomplish this goal, the rehosting process comprises several key components. The first component is the Virtual Environment (VE), which serves as a software environment that allows code execution with transparency, meaning that the execution can be observed in some capacity. The second component is the Hardware Emulation System (HES), which is a specialized VE designed to faithfully replicate the features of one or more targeted hardware components, essentially functioning as an emulator. The third component is the Rehosted Embedded System (RES), which combines a firmware image and a VE to adequately reproduce the hardware dependencies of the firmware. This ensures that the analysis conducted on the RES generates results that accurately reflect its original hardware environment [6].

Ideally, for the purpose of fuzzing, a VE running on an HES is preferred. However, HESs provide an exact representation of the embedded device, limiting their compatibility with various devices. Therefore, the RES offers a more focused approach by implementing only the critical hardware functions necessary for comprehensive system analysis. Given that this aligns precisely with the problem faced in the design of fuzzing frameworks, the RES becomes the central focus of the rehosting process [6].

Consequently, the rehosting process is defined as the construction of an RES for a specific embedded system, enabling the execution of a designated dynamic analysis task, such as fuzzing. Furthermore, it is established that once the firmware can execute within a VE, it satisfies the requirements for fuzzing [6].

Another essential aspect of building accurate VEs is understanding the modifications made to various abstraction layers, namely the function, application, operating system (OS), and hardware layers. These layers can be adjusted to facilitate the rehosting process. It should be noted that vulnerabilities can arise from bugs present in a single layer or multiple layers, as well as from interactions between layers. To address this, the research paper introduces EmbedFuzz, which utilizes high-level modeling to represent hardware peripherals as software libraries. This approach allows for the observation of interactions between layers [6].

Furthermore, due to variations in Instruction Set Architectures (ISAs), even within the same architecture (e.g., 64/32-bit, ARM/Thumb/ThumbV2), it becomes necessary to have a mechanism to interpret instructions specific to a particular ISA within a VE. This mechanism is referred to as a Virtual Execution Environment (VXE). It is worth noting that an HES provides an accurate and detailed reconstruction of the original hardware, requiring a VXE that fully captures the semantics of the ISA. On the other hand, an RES only needs to support the subset of the ISA used by the firmware [6].

Lastly, accurately modeling peripherals within VEs is crucial because they often serve as sources of attacker-controlled code. Different approaches can be employed to implement this behavior. For instance, Hardware-in-the-Loop forwards device interactions to the actual hardware, but this method requires debugging access to the original execution environment and faces scalability issues. Symbolic Abstraction, on the other hand, considers all values obtained from hardware as symbolic, integrating them into dynamic analysis. This approach requires the VXE to possess symbolic capabilities to interpret the hardware output. However, symbolic execution is prone to state-space explosion, necessitating a close approximation of the possible hardware outputs [6].

Recall that the main challenge we are facing is that different types of embedded systems differ vastly in their rehosting complexity. Depending on the type of system we will also have to find a method of firmware extraction, since not all firmware is publicly available [6]. This introduces another variable of firmware extraction which we omit here as it out of scope for this project.

2.3 Fuzzing Frameworks

We briefly introduce the three frameworks that we will be examining in this paper. Fuzzware [10], P2IM [7] and EmbedFuzz [8] each introduce novel concepts that contribute greatly to the efficiency and scalability of fuzzing frameworks. We have selected these three frameworks due to each having contributed to different challenges, which we will discuss in section 4.1.

2.3.1 Fuzzware

Fuzzware aims to efficiently explore firmware behavior while reducing input overhead. It achieves this by leveraging lightweight program analysis techniques and dynamic symbolic execution (DSE) to model firmware code behavior and generate constraints representing all possible uses of hardware-generated values. By evaluating these constraints, Fuzzware effectively narrows down the values to be explored [10].

The design of Fuzzware consists of an emulator and a coverage-guided fuzzing engine. The emulator enables the modeling of firmware code behavior, while the fuzzing engine drives the

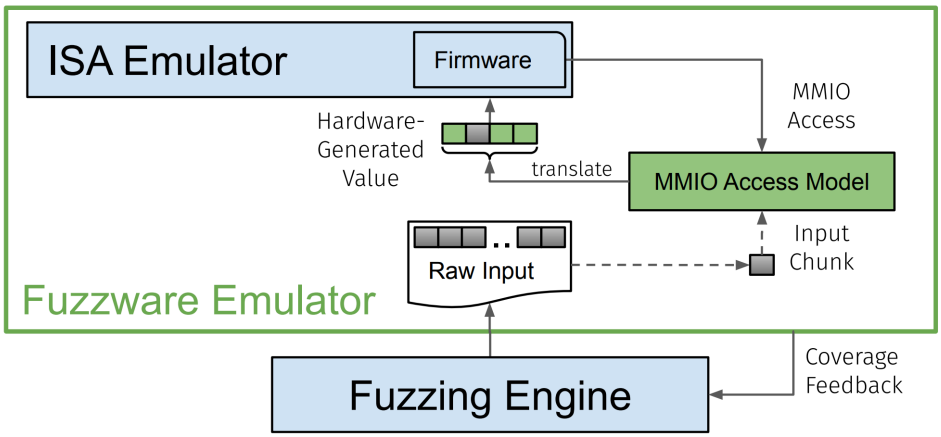

Figure 2.1: Figure 4 in Fuzzware [10] by Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi.

exploration process. The framework considers various aspects such as error handling, execution stalls, and the fuzzing loop, which contribute to the effectiveness and reliability of the fuzzing process [10].

A key component of Fuzzware is the modeling approach for memory-mapped I/O (MMIO) accesses. It defines five generic MMIO access models that handle input overhead efficiently. These models cover scenarios such as constant values, passthrough accesses, bitmask operations, set-based control flow, and full access with all bits utilized. By preserving firmware code paths and enabling reproducible translations, these models facilitate a comprehensive and systematic exploration of firmware behavior [10].

Fuzzware represents a significant contribution to firmware fuzzing by introducing an efficient approach that minimizes input overhead. Through its lightweight program analysis techniques and dynamic symbolic execution, it offers improved security and robustness in firmware systems. By reducing the exploration space and efficiently modeling firmware behavior, Fuzzware opens avenues for enhanced vulnerability detection and mitigation in the firmware domain [10].

2.3.2 P2IM

P2IM [7] presents a comprehensive framework designed to support fuzzers as drop-in components, enabling scalable and hardware-independent firmware testing. By bridging the gap between fuzzers and firmware, the framework allows fuzzers to focus on generating inputs and improving bug detection without requiring in-depth knowledge of the underlying software and hardware design of the MCU. Instead of relying on hardware or peripheral emulation, the framework utilizes approximate MCU emulation by automatically generating emulators based on firmware binaries. This approach

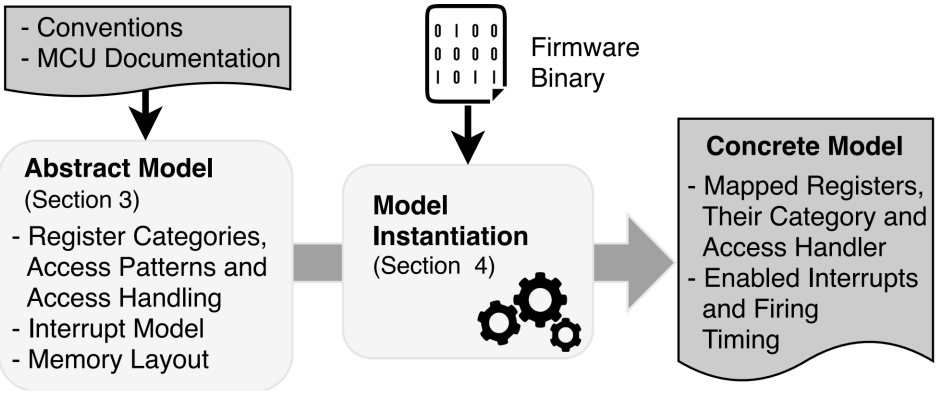

Figure 2.2: Figure 3 in P2IM [7] by Bo Feng, Alejandro Mera, and Long Lu

comprehensively exercises the firmware and facilitates various firmware analyses, such as taint analysis and invariant detection, without compromising functional correctness [7].

A key aspect of the framework's design is the definition of the Processor-Peripheral Interface Equivalence (P2IE) property for MCU emulators. This property ensures that the emulated interfaces are equivalent to the expected peripherals in terms of their impact on firmware execution. The framework includes abstract models for MCU architectures and employs automatic model instance for each firmware being tested. Moreover, the framework seamlessly integrates fuzzers as black-box input generators, feeding fuzzing inputs to the firmware through the emulator's peripheral interface access handlers. Additionally, the framework provides standard coverage feedback to the fuzzers, collected via the emulator [7].

2.3.3 EmbedFuzz

EmbedFuzz [8], is designed to efficiently and accurately fuzz embedded firmware by employing transplantation and High-level Modeling (HLM) techniques. In the absence of a classic operating system, EmbedFuzz targets Arm Cortex-M MCU firmware and transplants it into an executable user-space binary. This enables fast and native execution on Cortex-A processors, addressing performance overheads and ensuring correct instruction execution on real hardware. During the binary preparation phase, ISA mapping, coverage instrumentation, and peripheral emulation code insertion are performed. The runtime environment of EmbedFuzz plays a crucial role, handling complex instruction emulation and peripheral behavior at the HAL layer instead of the I/O instruction level. This approach allows for the flexible replacement of hardware peripherals with software implementations, leveraging existing handlers from HALucinator [4]. Furthermore, the run-time accurately and reproducibly emulates interrupts, crucial for effective bug triaging. Employing an interrupt scheduling approach similar to P2IM, EmbedFuzz delivers interrupts in a round-robin fashion based on a basic-block counter-based clock. Hence EmbedFuzz facilitates efficient and accurate fuzzing

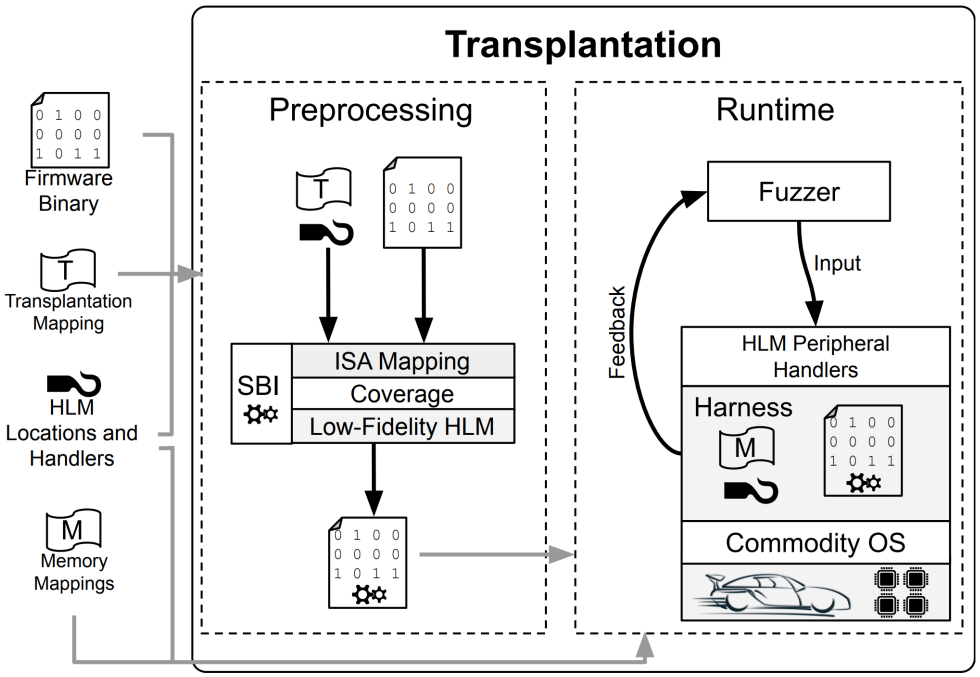

Figure 2.3: Figure 1 in EmbedFuzz [8] by Florian Hofhammer et al.

of embedded firmware with high performance and reliable peripheral emulation [8].

Chapter 3

Design comparison

In this chapter we will state challenges and compare how proposed state-of-the-art emulators overcome them. In chapter 4 we will then discuss how these solutions scale when the architecture support is extended, as all three frameworks currently focus on ARM Cortex-M.

3.1 Scalability

As fuzzing and dynamic analysis require many runs to produce meaningful results, being able to scale frameworks horizontally, meaning concurrent execution, is highly desireable. Approaches such as Hardware-in-the-Loop are usually not suited for this, as every run requires access to an isolated hardware unit [6].

Fuzzware: Recall from subsection 2.3.1 that Fuzzwares main goal is reduction of input overhead for MMIO modeling [10]. Although the original paper does not directly state how its novel access modeling directly affects scalability, we can infer that the reduction of input overheads translates into into faster fuzzing runs. This is due to the firmware querying more data then it needed from MMIO devices, leading to potentially irrelevant fuzzing runs in which non critical input values are fuzzed [10]. We refer to section 3.4 for a more detailed explanation of access models.

P2IM: P2IM implements a scalable approach that does not rely on specific hardware during dynamic analysis. Moreover thanks to the abstract model used for peripherals, reusing an instance of a generic model is assumed to be possible, increasing the performance across concurrent fuzzing runs when branching horizontally [7]. We will elaborate on how this concept works in section 3.4. For now our main takeaway should be, that the scalability of P2IM does not directly rely on any hardware specific considerations, but may be influenced by assumptions made about the abstract model [7].

EmbedFuzz: The novel idea behind EmbedFuzz, transplanting the firmware onto a high-end CPU that supports a similar ISA, makes it very scalable due to having no emulation costs [8]. As an example we are able to run and fuzz any Cortex-M firmware on a Cortex-A chip, using the transplanting technique described, which we will elaborate on in section 3.3. Note that eventhough this does introduce a hardware dependency, since we still require access to a high end CPU of the same architecture, the overhead that would be introduced by emulating a foreign ISA significantly outweights that [8]. Moreover just one high-end chip is execute many different fuzzing tasks for the firmware at the same time, as it executes every run in virtual memory.

3.2 Memory representaion

Mapping firmware specific addresses during rehosting is a delicate process, with the potential to introduce new bugs due to static addressing in the firmware binary. Therefore any information provided about the MCUs specific memory layout is crucial in being able to rehost binaries successfully [6].

Fuzzware: As with nearly all emulators examined in this paper, Fuzzwareassumes RAM ranges and MMIO mappings to be given by the MCU documentation. As long as this is provided, Fuzzwareis able to generate an access model for the MMIO accesses, which pose its main contribution to the fuzzing framework [10]. We will discuss these in section 3.4.

P2IM: Similar to Fuzzware, we again assume that sections such as flash, RAM and MMIO are provided to us by the MCU documentation. Additionally, this information is necessary in order for P2IM to derive an abstract model, which then firstly detects MMIO accesses and secondly can later derive the type of access from the access pattern to a particular register [7]. Please refer to section 3.4 for a more detailed explanation.

EmbedFuzz: Due to natively executing firmware after transplantation, EmbedFuzz utilizes the hosts MMU to mimic the physical address space of the MCU in virtual memory. The process isolation provided by virtual memory is another reason why transplantation is such a powerful technique. However to fully achieve this remapping into virtual memory, as with the other frameworks, we still require prior knowledge of the MCUs memory map [8].

3.3 VXE modeling

In section 2.2 we talked about the concept of an RSE only implementing firmware critical parts of the underlying ISA. Hence it is important to see how the different emulators choose to implement a VXE and how they decide which parts of the underlying ISA to implement.

Fuzzware: To support a wide range of different ISAs and focus on the task of rehosting itself, Fuzzware outsources the job of emulation itself to Unicorn Engine. Unicorn utilizes a just-in-time compilation (JIT) approach to efficiently translate and execute machine code on different processor architectures. This allows emulating and studying software across multiple platforms without requiring physical hardware or modifying the original code [5]. However as this is happens fully in software there is a lot of overhead, since Fuzzwaredoes also not modify the original instructions used by the firmware [10].

P2IM: Similiar to Fuzzware, P2IM outsources the task of emulation to QEMU. QEMU (Quick Emulator) is a widely used open-source virtualization tool that provides features such as hardware virtualization support, device emulation, and snapshot functionality [2]. The latter is an integral part of how P2IM performs its abstract model instance for MMIO accesses [7], which we will discuss in section 3.4. One adaption made is how interrupt firing is handled in P2IM, as the interrupt controller provided by QEMU does not support MCU specific interrupts. Thus the current prototype of P2IM uses a simple interrupt firing strategy, where enabled interrupts are fired in a round-robin fashion at a fixed interval [7].

EmbedFuzz: One of EmbedFuzz's novel ideas is its way of handling the ISA rehosting process. It proposes transplantation of the firmware binary from an embedded MCU to a high-end chip implementing the same general CPU architecture [8]. The provided example focuses on Cortex-M and Cortex-A chips, both of which are instances of the ARM architecture. However, as they are both different instances, a subset of their instructions differ slightly in semantics, which makes direct transplantation impractical. Therefore EmbedFuzz uses a rewriting step, during which the Cortex-M specific instructions are identified and replaced with the semantic equivalent in the Cortex-A ISA. As discussed in section 3.1, this enabled EmbedFuzz to eliminate the emulation overhead present in the other frameworks during execution. On the other hand, it does require access to a high performance chip of the firmware architecture [8].

3.4 Peripheral model and handeling

Fuzzware: With Fuzzware we aim to minimize input overhead in fuzzing by introducing access models. It operates by loading a firmware image into Unicorn Engine and setting up a harness to intercept MMIO accesses. The harness uses a raw input generated by a coverage-guided fuzzer to serve MMIO accesses, either by directly using a model or by translating a chunk of raw input into a hardware-generated value [10]. The framework runs the firmware code until the raw input is exhausted, and coverage feedback is reported to the fuzzer. Meanwhile, Fuzzware dynamically generates models for newly encountered MMIO accesses using symbolic execution. This self-adapting firmware emulation environment allows for effective exploration of unknown firmware with reduced input overhead, providing a generic and adaptable framework for fuzzing [10].

P2IM: The P2IM framework introduces another approach to firmware testing and fuzzing without the need for MCU devices, hardware peripherals, or human intervention. It employs Processor-Peripheral Interface Modeling (P2IM) to derive models for processor-peripheral interfaces, focusing on on-chip peripherals. The modeling process consists of two steps. Firstly, an abstract model is manually defined, capturing generic patterns, conventions, and acceptable input for accessing processor-peripheral interfaces. This abstract model serves as a template for different MCU architectures. Secondly, model instantiation automatically incorporates firmware-specific information, such as memory mappings and inter-dependencies, necessary for accurate emulation. The framework utilizes explorative firmware executions to infer this firmware-specific information [7]. In addition, the P2IM framework seamlessly integrates fuzzers as input generators, directing fuzzing inputs to the peripheral interface access handlers in the QEMU emulator. Standard coverage feedback is provided to the fuzzers through the emulator, facilitating effective fuzzing. By focusing on interface equivalence rather than peripheral emulation, P2IM enables hardware-independent, scalable, and high-coverage firmware testing [7].

EmbedFuzz: For EmbedFuzz, the run-time handles the emulation of complex instructions and peripheral behavior, serving as a fuzzing harness for integration with general-purpose fuzzers. Peripheral interactions are tackled at the higher-level HAL, a software library that abstracts peripherals into callable interface functions [4] rather than at the instruction level, allowing for the replacement of hardware peripherals with software implementations. The run-time interfaces with the prepared binary, intercepting calls to the firmware's HAL and providing emulation code for each HAL function. The run-time also ensures reproducible emulation of interrupts, employing an interrupt scheduling approach similar to P2IM [7] and HALucinator [7]. This approach enables regular interrupt simulations, such as a timer interrupting the firmware to poll an attached peripheral. By leveraging these design decisions, EmbedFuzz enables effective fuzzing of embedded firmware with accurate emulation of complex instructions and peripheral behavior [8].

Chapter 4

Extending architecture support

As all of the emulators discussed in chapter 3 focus on and implement Cortex-M support in practice, our goal is to survey and state potential issues that may arise while extending the set of supported architectures. Furthermore, in chapter 5, we will provide an explanation of how we added partial RISC-V support to EmbedFuzz.

4.1 Challenges

First let us define the main challenges we have identified while surveying the different frameworks.

4.1.1 Memory representation

Depending on how narrow the specifications for a particular architecture are defines how dynamic an RES would have to be in its underlying implementation. To elaborate, if the memory addresses of MMIO, RAM and flash are unknown by default, as architectures do not always strictly specify them, then it would be difficult for all examined frameworks to operate on that architecture. This is due to the assumption of prior knowledge of these regions, as stated in section 4.1. Subsequently, if the RES is developed with current popular architectures for MCUs in mind, it might not adapt well to newly emerging ones.

For instance, RISC-V is much less specific in terms of memory management and layout in its official specification, allowing for a very diverse spectrum of chips, with different memory regions and interrupt handlers to exist [1].

In contrast ARMv7 is more strict and clearly defines a basic memory map that all implementations will follow, hence making it easier to implement an RSE for all such chips [9]. Furthermore all

interrupts across the Cortex-M line are handled in a similar fashion and only minimal adaptations are need to support further ARM chips, should the need arise [9].

4.1.2 Intra-architecture specific ISAs

Recall from section 3.3 that we talked about the necessity of a VXE within an VE to interpret instructions for a given ISA. This on its own poses a challenge for choosing which features of an instruction set to support for a specific ISA, and is thus likely amplified when ISAs differ within the same architecture. Moreover, as we saw with EmbedFuzz in section 3.4, the side effects of instructions may differ

For instance, take the difference in RISC-V 64bit and RISC-V 32bit, which boils down to differing arithmetic instructions, where the conversion between them requires flipping one bit in the opcode [1]. Hence to add full RISC-V support we would need to support 32bit and 64bit seperately.

Going back to section 3.3 we see that this is not necessarily an issue for Fuzzwareand P2IM, as both of them use full software emulation, meaning the emulator needs to support the new ISA. Since both Unicorn Engine [5] and QEMU [2] offer a wide range of ISA support, this is an advantage compared to EmbedFuzz, which requires a rewriting step during transplantation to facilitate firmware execution for architectures like ARM Cortex-M on Cortex-A chips [8].

4.1.3 Interrupt handling

Recall that interrupts are a way of interacting with peripherals and that as discussed in chapter 2, are a likely source of vulnerabilities. Handling interrupts correctly and facilitating a sufficient interface to make dynamic analysis possible therefore poses a challenge when interrupts are handled differently from intra- and/or inter-architecturally. In section 3.2 we already briefly talked about how interrupt handling might differ depending on the instantiation of the specification.

As an example, RISCV-V and Cortex-M different ways of handling interrupts. Whereas the latter is more strict in its specification of vector table and interrupt handling using its nested vectored interrupt controller (NVIC) [9], while the former allows for some leniency in implementing different interrupt handlers and how vector tables are used [1].

From section 3.3, we know that P2IM acknowledges the importance of emulating

4.2 Further considerations

The following points are not necessarily framework specific but cover a broader issues that relate to resources and tools needed to implement these frameworks.

4.2.1 MCU documentation availability

We previously stated in subsection 4.1.1 that MCU designs are often propieteray and depending on architecture and specification, might differ vastly from one another. Since all of our frameworks require prior knowledge of some parts of the MCU design, see section 3.2 and subsection 4.1.3, obtaining specific MCU documentation is a crucial task.

4.2.2 Toolchain support

Recall from section 2.2 that for the purpose of fuzzing we aim to achieve RES level emulation. Hence it is necessary for our architecture and ISA to have sufficient support within the ecosystem that is used to build and execute such firmware. For the case of EmbedFuzz, we heavily rely on the availability of compiler toolchains for cross compilation of firmware as part of our VXE as well as the availability of certain libraries [8]. However this might not be the case for all firmware, and as we will see in chapter 5, architectures such as RISCV-32bit, which currently has Linux kernel support but is missing crucial libraries for 32bit, complicating the creation of dynamic intra-architecture support.

Chapter 5

Implementation

As part of this paper we extend EmbedFuzz with initial support for partial RISCV-V firmware loading. We will refer to the challenges stated in section 4.1 how they were dealt with.

Starting with the toolchain support, the initial build architecture of EmbedFuzz only supported ARM Cortex-M. Overhauling the build architecture mainly revolved around adapting the docker-containers used for building, running and later rewriting from the previous ARMv7 implementation to instead use RISC-V. Furthermore the toolchains used for building were changed from ARM to RISC-V. This is also were we first encountered our first problem, as the used RISC-V 64bit GNU GCC toolchain does not support cross compilation for RISC-V 32bit and several packages needed for the setup, such as Python 3.10, had no support for RISC-V 32bit under Linux. Therefore, when choosing which RISC-V ISA to implement for initial support we stuck with RISC-V 64bit, as the provided toolchain support was much better.

Next, numerous adaptations were necessary to those files that included architecture specific addresses as well as code. Moreover most of these were ARM specific bit masks, which are not needed for RISC-V 64bit as there is no need to preserve the condition flags [1].

Subsequently, the assembly files responsible for mapping the general purpose registers were modified to now support RISC-V 64bit. Note that this distinction is important as the ISA differs for RISC-V 32bit for some arithmetic instructions [1].

Afterwards, we modified the loader and its linker file to support 64bit Elf files and RISC-V 64bit respectively. To this end we used the default linker script provided by GNUlibc, for RISC-V 64bit, with minor modifications to the mapping of the text segment. These modifications had to be made to ensure that the loader text segment would not interfere with the firmwares text segment during loading of the firmware binary. As an example, by default both the loader and firmware binary have a start function that initiates execution and later jumps to main [1]. If we expect the firmware to be compiled with the default GNUlibc linker script, our loader cant use the address in virtual memory

for its text segment as firmware, since the firmware is executed inside the loader, and hence in the same virtual address space. Therefore ,i.e., both start functions would overlap, possibly causing undefined behavior in the loader.

Lastly, as we were not able to complete full RISCV-V support due to time constraints, we briefly talk about future considerations for adding achieving full support. Beginning with the differences between RISC-V 64bit and 32bit, transplantation support for RISCV-V 64bit to 32bit would need to be added to the rewriter. This can be done by flipping one bit in the instruction opcode of RISC-32bit instructions [1]. When it comes to toolchain support for 32bit, it would certainly be possible to modify and build the required packages but was omitted here due to time constraints.

Chapter 6

Conclusion

In summary, after examining several state-of-the-art fuzzing frameworks and their solutions to challenges discussed in section 4.1, we were able to gain insights into the current requirements for architecture specific resources needed to extend the set of those supported by the framework. Non-specificity in defining memory maps as well as interrupt handling and firing may thus lead to manual overhead when adding support for such architectures. Additionally, we also identified issues specific to the underlying support of MCU architectures on general purpose CPUs in chapter 5.

Bibliography

[1] Krste Asanovi Andrew Waterman1 and John Hause. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Dec. 2021. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

[2] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506. DOI: 10.1109/TSE.2017.2785841.

[4] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1201–1218. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

[5] Hoang-Vu Dang and Anh-Quynh Nguyen. "Unicorn: Next Generation CPU Emulator Framework". In: Jan. 2015.

[6] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. "SoK: Enabling Security Analyses of Embedded Systems via Rehosting". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 687–701. ISBN: 9781450382878. DOI: 10.1145/3433210.3453093. URL: https://doi.org/10.1145/3433210.3453093.

[7] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1237–1254. ISBN: 978-1-939133-

17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[8] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1237–1254. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng.

[9] ARM Joseph Yiu Senior Embedded Technology Manager. *Cortex-M for Beginners*. May 2017. URL: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2057-00-00-00-01-28-35/Cortex_2D00_M-for-Beginners-_2D00_-2017_5F00_EN_5F00_v2.pdf.

[10] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.