École Polytechnique Fédérale de Lausanne

Exploration and Enhancement of Firmware Emulation in the Security Industry Context

by Duo Xu

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Patrik Marcacci
Thesis Supervisor

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

September 17, 2024

Dedicated to my family.

Acknowledgments

The past three years have been the most physically and mentally demanding period of my life. Confronting and managing my long-hidden anxiety disorder and stomach issues while completing my studies was incredibly challenging, especially during the final months of my Master's project. Fortunately, I have come through it all. My mindset is now stronger than ever, and my physical health continues to improve.

As I reflect on my time in Lausanne, I am deeply grateful to Prof. Mathias Payer, who introduced me to the field of embedded systems and generously provided me with the opportunity to work as a lab assistant. His kindness and professionalism have been a source of profound inspiration. I also wish to extend my heartfelt thanks to my lab mates in Hexhive. Their invaluable support and the passionate discussions we shared about research greatly enriched my academic journey.

I am also immensely thankful for the unwavering love and support of my parents, and for Yiwen's companionship during my most difficult times. Their encouragement has been the anchor of my life. Additionally, I would like to thank Patrik and Stephane for their consideration and understanding of my stomach issues during my internship.

Lastly, I want to thank myself for not giving up. It was through perseverance that I was able to confront and overcome these challenges, ultimately strengthening my mindset and emerging more resilient.

Lausanne, September 17, 2024 Duo Xu

Abstract

Embedded devices have become ubiquitous in our daily lives, making the analysis of their security increasingly important. Traditional methods of manual analysis on actual hardware are limited by their low inspection and post-mortem capabilities. Recent research has proposed fuzz testing firmware within emulated environments; however, existing solutions only support the Cortex-M architecture. In practice, many analysis targets use the Cortex-A architecture, rendering these approaches insufficient for broader application.

In this work, we select Fuzzware as our study target to examine its scalability and usability. We design and implement a Cortex-A (ARM32) extension for Fuzzware. Additionally, we develop a GDBServer within Fuzzware to enhance its dynamic analysis capabilities and explore a method for automatically synthesizing peripheral handler functions. We validate our Cortex-A implementation by exploiting a buffer overflow vulnerability in the Tegra X1 BootROM. Our findings suggest that Fuzzware is more suited for monolithic firmware that does not rely on advanced hardware features. For analyzing complex firmware with an operating system, extending architecture support on Hoedur could be a more suitable approach since it uses QEMU as the base emulator.

Contents

Chapter 1

Introduction

The rapid development of embedded devices, such as drones and smart home systems, has made them ubiquitous in daily life. This widespread presence has heightened the importance of security for IoT devices. Security engineers are often required to conduct reliability analyses of IoT devices to identify potential vulnerabilities in some products. However, several challenges posed by physical testbeds can significantly reduce analysis efficiency. Firstly, physical devices typically offer limited introspection capabilities. Additionally, when a crash occurs, it takes time for the device to restart and reach the crash site, which greatly hinders the efficiency of post-mortem analysis. Furthermore, due to the lack of hardware dependency support, dynamic analysis techniques like fuzzing are difficult to apply. Beyond these issues, the manual analysis of embedded firmware is inherently complex and time-consuming, requiring the analyst to possess a thorough understanding of the system under test.

In response to these challenges, researchers have begun to explore the possibility of conducting automated fuzz testing in an emulated environment. Works like FIRMADYNE [3] and Firmfuzz [22] allow researchers to dynamically analyze Linux-based firmware. To further support emulating monolithic firmware, modern emulation systems such as QEMU [1] and SIMICS [16] are utilized. However, emulation on such systems requires non-trivial efforts to manually craft software equivalents of the peripherals needed by the firmware.

Recent approaches have increasingly focused on addressing the need for hardware access in an automated manner, leading to three main lines of research. First, several methods [2, 20, 27] leverage symbolic execution, treating hardware registers as symbolic inputs. By solving the collected symbolic constraints along the execution path, they effectively model peripheral behaviors and ensure firmware execution. The second line of research [4, 15, 17] involves providing manually implemented handler functions to emulate peripheral logic. Lastly, other approaches [6, 8, 9] aim to model peripheral behaviors through pattern-based modeling, dynamically matching peripheral accesses with predefined patterns to identify static hardware register types (e.g., control registers).

Kudelski's security engineers aim to apply these state-of-the-art methods to their analysis of IoT products. However, there is a significant gap in directly implementing these solutions. First, current implementations are limited to supporting Cortex-M firmware, while most real-world analysis tasks involve products using the Cortex-A architecture (this paper considers only the ARM32 ISA). Moreover, the evaluation of these methods is largely confined to a set of benchmark firmware collected by P2IM [6]. This limitation raises concerns about the applicability and robustness of these systems when targeting other firmware.

Among the state-of-the-art solutions, we select Fuzzware [20] as our primary focus. Based on the experimental results presented in the literature, Fuzzware and Hoedur [21] demonstrate the best performance on the P2IM benchmark. While Hoedur generally surpasses Fuzzware in fuzzing capabilities, it employs a similar emulation design. Therefore, we choose to first explore Fuzzware and then consider Hoedur if necessary. A detailed comparison of the state-of-the-art approaches is provided in Table 1.1.

Approach	Description
HALucinator	Replace library functions with manually crafted handlers.
μ-Emu	Explore firmware logic through guided symbolic execution.
P2IM	Match register types via pattern-based modeling.
Fuzzware	Automatic modeling through coverage-guided fuzzing.
Hoedur	Firmware-aware fuzzing integration based on multi-stream inputs.

Table 1.1: Comparison of state-of-the-art rehosting works

In this work, we systematically explore the scalability and usability of Fuzzware. We carry out a case study of emulating and triggering the Fusée Gelée vulnerability to explore the extension of Fuzzware to the Cortex-A architecture. In the meantime, we support the GDBServer to enhance Fuzzware's post-mortem and dynamic analysis capability. Additionally, we came up with a design of automatical synthesis of peripheral handler function templates.

Chapter 2

Background

2.1 Monolithic Embedded Firmware

Firmware typically refers to the code running on embedded systems, encompassing both the logic of software applications and the management of hardware interactions. Depending on specific requirements, firmware may include a minimal OS or library functions to facilitate hardware communication. This study focuses on monolithic firmware, which lacks metadata similar to that found in traditional binaries, making its analysis significantly more challenging. Such scenarios are frequently encountered in the daily work of security engineers.

2.2 MCU Peripherals

A Microcontroller Unit (MCU) comprises one or more CPUs, along with memory and input/output peripherals. These peripherals, such as UART and GPIO, facilitate communication with the external world. Generally, there are three primary methods for the CPU to interact with these peripherals: memory-mapped I/O (MMIO), interrupts, and direct memory access (DMA).

In MMIO, peripherals are assigned specific memory regions containing MMIO registers. These registers are used to control peripheral behavior and facilitate data transfer. The CPU directly interacts with these peripherals using load/store instructions, requiring it to actively monitor the memory-mapped registers for changes.

Interrupts are asynchronous signals sent by hardware devices to the CPU to indicate that an event has occurred, such as data becoming available or an error condition. This method allows the CPU to continue executing other tasks without the need to continuously poll devices, improving overall efficiency by only handling the event when an interrupt is received.

DMA is a technique that enables devices to transfer data directly to or from system memory without CPU intervention. By offloading the data transfer task to the device, DMA frees up the CPU to handle other processing tasks. This is particularly useful in high-speed devices, such as disk drives and network cards, where rapid data transfer is essential.

2.3 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) is a program analysis technique that combines concrete execution (actual program runs with specific inputs) and symbolic execution [13](running the program with symbolic inputs to explore multiple execution paths). This hybrid approach is used primarily for software testing, bug finding, and security vulnerability analysis.

Symbolic execution is a technique where program inputs are treated as symbolic variables rather than concrete values. This allows the exploration of multiple execution paths in a program simultaneously. There are three key points in symbolic execution. The first one is path constraints. As the program executes, symbolic execution collects constraints on inputs that must be satisfied to follow a particular execution path (e.g., branch conditions in if-statements). The second is constraint solving. When the program encounters a branch statement (e.g., $x > 0$), the symbolic execution engine generates a condition for each path. The execution engine uses a constraint solver to determine if the conditions can be satisfied. The last one is path exploration. The symbolic execution engine will try to execute all possible paths to cover more program semantics, which is important for tasks like automated testing. However, having too many branches will result in path explosion and prevent the symbolic execution engine from analyzing complex programs. Utilizing both symbolic and concrete execution can help overcome the limitations of both methods.

2.4 Fuzzware

Fuzzware is a state-of-the-art rehosting system that employs the Unicorn Engine [19] as its base emulator and Angr [24] as the symbolic execution engine. An overview of the system is provided in Figure 2.1. When the emulated firmware attempts to access a peripheral register (designated as an MMIO region), Fuzzware takes a snapshot of the current program state. This snapshot is then loaded into Angr, where execution resumes from the point of the MMIO access. During this process, Fuzzware treats the register and stack values at the snapshot entry as symbolic. By analyzing the collected constraints along the execution path, Fuzzware determines if the current MMIO access conforms to one of the predefined models.

Once an appropriate model is assigned to the MMIO access, it assists in taking a chunk of fuzz input and translating it into an MMIO access value. The complete list of predefined models is detailed

Model	Description
Constant	MMIO access needs to return a constant.
Passthrough	MMIO access doesn't affect the execution.
Bitextract	Only part of the MMIO access value is used.
Set	MMIO access value should be within a discrete set.
Identity	None of the above, needs to offer fuzz input for full MMIO access.

Table 2.1: Fuzzware MMIO models

in Table 2.1. Among these models, the Constant and Set models are primarily used by Fuzzware to emulate peripheral semantics. They gather values from branch conditions associated with the current MMIO access. By constraining the MMIO access values to a specific constant or discrete set, the firmware can pass the necessary checks and proceed with execution. For instance, the Constant model might be assigned to an MMIO access of the UART status register. This read access must return `UART_HAS_DATA` to prevent the firmware from getting stuck in the status-checking logic. The remaining models are designed to enhance fuzzing efficiency.

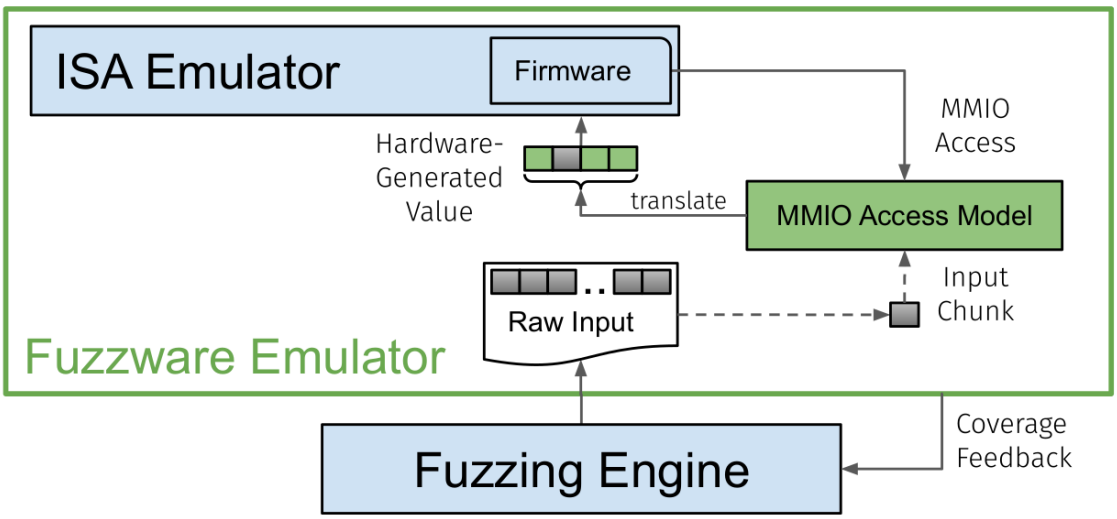

Figure 2.1: Fuzzware system overview

Chapter 3

Design

In this section, we outline the design decisions made for the Cortex-A extension on Fuzzware. To meet the company's requirements, the extension primarily targets monolithic firmware, such as BootROM and Bootloader. The section is divided into three parts, detailing the architectural differences between Cortex-M and Cortex-A. Following this, we discuss the design and implementation of the GDBServer, as well as our approach to automated handler synthesis.

3.1 Cortex-A vs. Cortex-M: Basics

Instruction Sets: Cortex-M and Cortex-A support three instruction sets: Thumb, Thumb-2, and ARM32. Thumb consists solely of 16-bit instructions, while Thumb-2 introduces some 32-bit instructions (e.g., MOV.W). In contrast, ARM32 exclusively uses 32-bit instructions. Cortex-M supports only Thumb and Thumb-2, whereas Cortex-A supports both Thumb/Thumb-2 and ARM32. This requires careful handling of instruction set switches during emulation for Cortex-A.

Registers: Both Cortex-M and Cortex-A share the same set of general-purpose registers (R0-R12, LR, PC, SP). However, they differ in their program status registers: Cortex-A uses CPSR, while Cortex-M uses xPSR. This register is essential for managing condition flags, interrupt handling, and more. Additionally, Cortex-A includes several banked registers (e.g., SP_IRQ) that preserve critical register states during mode switches.

Multi-core support: Cortex-M is limited to single-core execution, while Cortex-A typically supports multi-core configurations. Due to the limitations of Unicorn's emulation capabilities, our implementation only considers single-core scenarios.

Program entrypoint: Cortex-M firmware uses the first 8 bytes to store the program entry point and the initial value of the stack pointer. In contrast, Cortex-A has no such convention; the entry

address must be manually identified, and the initial stack pointer is set within the entry function.

3.2 Cortex-A vs. Cortex-M: Execution Modes

Execution mode is a crucial concept that defines the processor's execution state, determining the current privilege level, access rights, and exception-handling behavior. Table 3.1 provides an overview of the execution modes for Cortex-A and Cortex-M. In Cortex-M, the two modes distinguish whether the CPU is handling an exception. Given our focus on monolithic firmware, full support for all Cortex-A modes is unnecessary.

USER mode serves as the general execution state, requiring no additional consideration. The Interrupt Request (IRQ) mode, similar to Cortex-M's Nested Vectored Interrupt Controller (NVIC), is discussed in more detail in Section 4.3. SVC mode is entered when the firmware intentionally invokes the SVC instruction for system calls; this can be supported alongside interrupt handling. Fast Interrupt Request (FIQ) mode, which is simply an interrupt with higher priority, can be accommodated by assigning FIQ interrupts the highest priority.

Monitor (MON) mode is used for Cortex-A's TrustZone extensions. Since TrustZone is beyond our current scope, support for this feature is omitted, and any related logic in the firmware is bypassed if encountered. The Memory Management Unit (MMU) is typically used by an OS, which is not relevant for monolithic firmware. Therefore, support for the Abort mode is also unnecessary. Undef mode is triggered when the CPU encounters an undefined instruction. During emulation, an undefined instruction will crash the Unicorn engine and should be avoided or manually patched. System mode is disregarded as it is used by an operating system, which is outside the scope of our focus on monolithic firmware.

Arch	Mode	Description	Privileged
Cortex-A	USER	Unprivileged mode in which most applications run.	No
Cortex-A	FIQ	Entered on an FIQ interrupt exception.	Yes
Cortex-A	Supervisor	When a Supervisor Call instruction (SVC) is executed.	Yes
Cortex-A	IRQ	Entered on an IRQ interrupt exception.	Yes
Cortex-A	Monitor	Implemented with Security Extensions.	Yes
Cortex-A	Abort	Entered on a memory access exception.	Yes
Cortex-A	Hypervisor	Implemented with Virtualization Extensions.	Yes
Cortex-A	Undefined	Entered when an undefined instruction executed.	Yes
Cortex-A	System	Privileged mode, sharing the register view with User mode.	Yes
Cortex-M	Thread	Used to execute application software.	No
Cortex-M	Handler	Used to handle exceptions.	Yes

Table 3.1: Execution modes of Cortex-M and Cortex-A.

3.3 Cortex-A vs. Cortex-M: Interrupts Handling

The interrupt handling controllers in Cortex-M and Cortex-A are known as the Nested Vectored Interrupt Controller (NVIC) and the Generic Interrupt Controller (GIC), respectively. Under typical conditions, both NVIC and GIC follow similar procedures for managing nested interrupts. GIC, however, includes support for distributing interrupt handling across multiple cores, and it categorizes interrupts into three types: Software Generated Interrupts (SGI), Private Peripheral Interrupts (PPI), and Shared Peripheral Interrupts (SPI). Since our implementation supports only a single core, the GIC's distribution feature is not utilized.

A key distinction between NVIC and GIC is the location of the Interrupt Service Routine (ISR) table. In Cortex-M, the ISR table has a fixed address at 0x00000000, whereas GIC has no such fixed convention. As a result, the ISR table address for GIC must be manually configured within Fuzzware. Aside from these differences, our implementation of GIC can be largely based on the existing NVIC codebase.

3.4 GDBServer

A GDBServer can significantly enhance the dynamic analysis process. Currently, Fuzzware lacks a functional GDBServer implementation. During post-mortem analysis, analysts are limited to using Unicorn hooks to print executed blocks and record information throughout the execution. By contrast, a GDBServer allows analysts to set breakpoints dynamically at specific addresses and inspect register and memory values in real-time. The GDBServer must adhere to the GDB Remote Serial Protocol (RSP) [7], which defines the packet structure used during the execution of GDB commands. This setup enables connection to various debugging clients, such as the GDB client in IDA Pro.

3.5 Automatic Handler Synthesis

Manually replacing peripheral functions with crafted handlers is a core feature in some rehosting systems, such as HALucinator [4]. However, creating these handlers requires analysts to have an in-depth understanding of the peripheral's semantics, making the process both cumbersome and demanding. To address this challenge, we propose an experimental design for an automatic handler synthesis approach. While it is impractical to automatically generate handlers that replicate the exact logic of the original peripheral functions, our goal is to model the side effects of these functions instead.

A function can have three types of inputs: global variable reads, input parameters, and MMIO

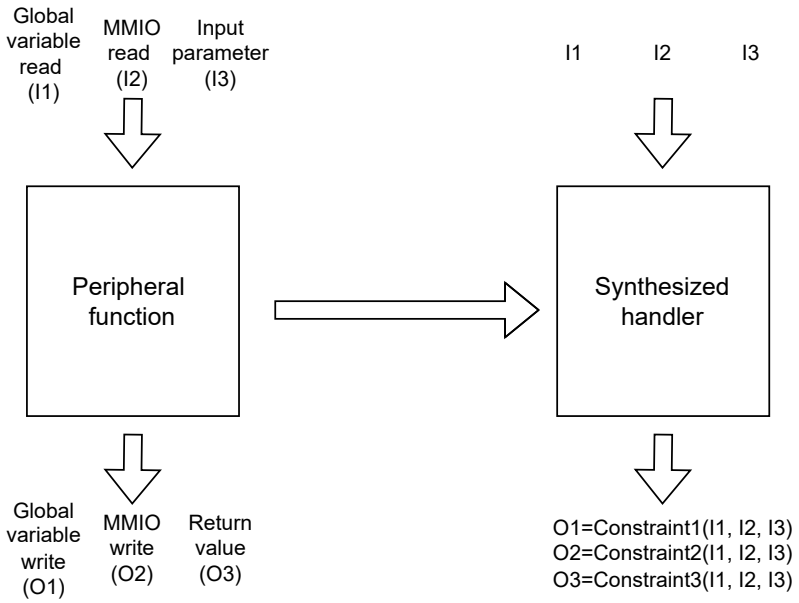

Figure 3.1: Simplified illustration of handler synthesis

reads. Correspondingly, there are three types of outputs: global variable writes, return values, and MMIO writes. By representing a function's output in terms of its input sources, we can approximate its logic. Inspired by Fuzzware's MMIO modeling, we leverage symbolic execution to facilitate our handler synthesis process.

Our approach begins with identifying the peripheral functions to be replaced. During emulation, whenever one of these functions is encountered, we take a snapshot of the current state. This snapshot is then loaded into Angr, with the input parameters recorded as symbolic variables. During symbolic execution, any reads from MMIO regions or global variables are noted as symbolic values. For each output (MMIO write, global variable write, or function return), we collect the symbolic constraints associated with these output variables. Ultimately, we represent the side effects of the original peripheral functions using symbolic constraints derived from the function inputs.

A simplified illustration of this process is shown in Figure 3.1, where I1, 2, 3 represent the three input sources and O1, 2, 3 represent the outputs.

Chapter 4

Implementation

4.1 Cortex-A Extension

The previous section provided a detailed discussion on extending Cortex-A support in Fuzzware. Here, we will highlight specific implementation aspects that require additional attention.

First, it is crucial to ensure that both Unicorn and Angr accurately emulate the instructions of the specified architecture. Angr's backend translates each instruction into Vex Intermediate Representation (IR) and emulates the firmware based on the behavior described in the IR. An example of this translation is shown in Figure 4.1. If Angr fails to translate a particular instruction, we must manually define the semantics of that instruction in Vex IR and replace the original instruction accordingly. Similarly, if Unicorn does not support certain instructions, we need to hook the instruction address using a `UNICORN_HOOK_CODE` handler. Whenever the emulator runs into this address, it uses our crafted handling logic and skip the unsupported instruction.

In addition to instruction support, it is essential to ensure that Angr correctly restores the program state from snapshots. For instance, Angr does not directly use the CPSR register as a reference for the program state. Instead, it relies on a `Code Condition Dependency` register to calculate the program state. Therefore, when loading the correct CPSR value from the snapshot, we must also configure the `Code Condition Dependency` registers to ensure Angr performs the program state calculations as expected.

Another important aspect is the potential flaws in Fuzzware's MMIO analysis implementation, which may lead to misclassification of some MMIO models. For example, the MMIO modeling function tracks variables read or written on the stack to record the liveness of each symbolic variable. This tracking is based on whether the current variable read or write operation involves the SP register. However, in some scenarios, the current stack base may be assigned to another register (e.g., R6). In such cases, the MMIO modeling logic fails to track the variable, leading to an incorrect model

assignment. To address this, we must manually inspect the generated MMIO models and correct any errors in the `config.yml` file.

```
00 | ------ IMark(0x825c, 4, 0) ------
01 | t10 = GET:I32(r11)
02 | t9 = Sub32(t10,0x0000000c)
03 | t2 = GET:I32(r3)
04 | STle(t9) = t2
05 | PUT(pc) = 0x00008260
```

```
STR     R3,
[R11,#var_C]
```

Figure 4.1: Example of Vex IR used by Angr

4.2 GDBServer

The primary function of the GDBServer is to manage breakpoints, which are essential not only for setting and clearing breakpoints but also for other execution control commands. For instance, the implementation of the `step into` command involves setting a temporary breakpoint at the next instruction, continuing execution, and then halting the emulator once this breakpoint is reached, after which the temporary breakpoint is cleared.

To implement this, we use a `UNICORN_HOOK_CODE` handler, a callback function that is executed on every instruction. This handler allows us to control breakpoint behavior by maintaining a list of breakpoints and checking each instruction's address to see if it matches an entry in the list. Currently, our GDBServer implementation supports commands such as `step into`, `step over`, `continue`, and `breakpoint add/clear`. An overview of our implementation is shown in Figure 4.2

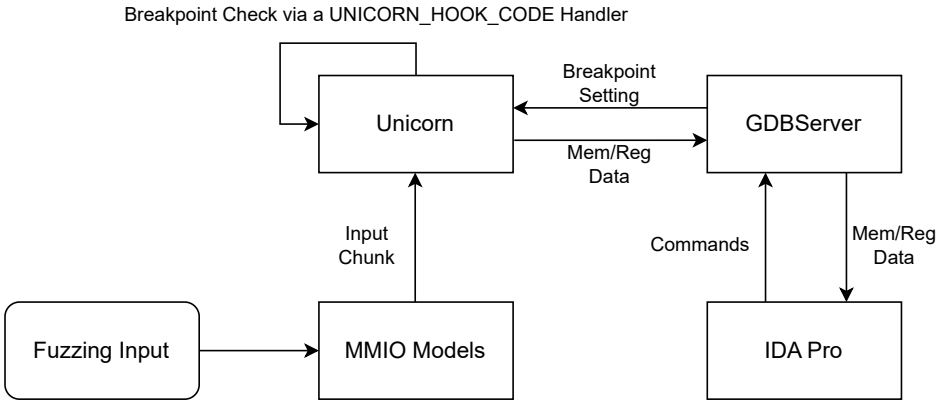

Figure 4.2: Overview of the GDBServer

Event	Reason
reg_read	Track initial read of each register after function starts.
reg_write	Record register write to R0 for return value.
mem_read	Track memory read outside of current stack frame.
mem_write	Record memory write.
exit	Stop side-effects recording if current function returns.
address_concretization	Check if the current concretized symbolic address is in the MMIO region.

Table 4.1: Angr events inspected for handler synthesis

4.3 Automatic Handler Synthesis

The primary challenge in implementing automatic handler synthesis lies in accurately identifying the three sources of function input. For input parameters, we adhere to the ARM parameter passing convention, where the first four inputs are passed via registers R0-R3, and any additional inputs are passed through the stack. Since symbolic execution begins at the start of the function, the starting point of the current stack frame is known. By monitoring memory read accesses outside of this stack frame, we can identify and track all three input sources effectively.

Angr's event callback feature is instrumental in this process, allowing us to inspect various aspects of the symbolic execution. The specific events we monitor and the reasons for their inspection are detailed in Table 4.1.

Chapter 5

Evaluation

We perform a case study of triggering the Fusée Gelée vulnerability in the Tegra X1's BootROM, to showcase our support for the ARM Cortex-A architecture.

5.1 Fusée Gelée

The Fusée Gelée vulnerability is a cold-boot exploit present in the Nvidia Tegra X1 BootROM. This exploit leverages a buffer overflow vulnerability in the CPU's USB Recovery Mode (RCM), allowing an attacker to bypass the Secure Boot process and execute arbitrary code. Since this vulnerability resides in the BootROM, it cannot be patched through software updates, making it more severe than typical software vulnerabilities.

To exploit this buffer overflow on a real device, such as the Nintendo Switch, the Tegra SoC must be manually forced into USB Recovery Mode. A simplified pseudo-code representation of the USB RCM logic is provided in Listing 5.1. After the USB successfully transmits the device ID via endpoint 1, the connection is established, and the system awaits the RCM command and payload on the same endpoint. While processing RCM commands, the firmware must also handle USB control requests on endpoint 0, such as checking the device status.

The buffer overflow occurs within the control request processing function, as shown in Listing 5.2. When the USB host processes a control request to check an endpoint's status, it uses the length specified in the request packet rather than relying on the `sizeof` function. The result of this control request is then copied into one of the USB DMA buffers (0x40009000 or 0x40005000). By specifying an excessively large length value, an attacker can corrupt the stack frame. With a carefully crafted RCM payload, the program flow can be redirected to execute the attacker's code. Figure 5.1 provides an illustration of this overflow process.

```
while (1) {
  usb_ops->handle_control_requests(current_dma_buffer);

  if (rcm_send_device_id() == USB_NOT_CONFIGURED) {
    usb_initialized = 0;
  } else {
    usb_initialized = 1;

    rcm_read_command_and_payload();

    rc = rcm_validate_command();
    if (rc != VALIDATION_PASS) {
      return rc;
    }

    rcm_handle_command_complete(...);
  }
}
```

Listing 5.1: USB RCM logic

```
void *data_to_tx;

uint16_t size_to_tx   = 0;

uint16_t length_read  = setup_packet.length;

if (setup_packet.request == REQUEST_GET_STATUS) {
  if (setup_packet.recipient == RECIPIENT_DEVICE) {
      status     = get_usb_device_status();
      size_to_tx = sizeof(status);
  }
  else if (setup_packet.recipient == RECIPIENT_ENDPOINT) {
      status     = get_usb_endpoint_status(setup_packet.index);
      size_to_tx = length_read;
  }
  else {
    // Handle other cases
  }
}

memcpy(dma_buffer, data_to_tx, size_to_tx);
```

Listing 5.2: USB control request handling

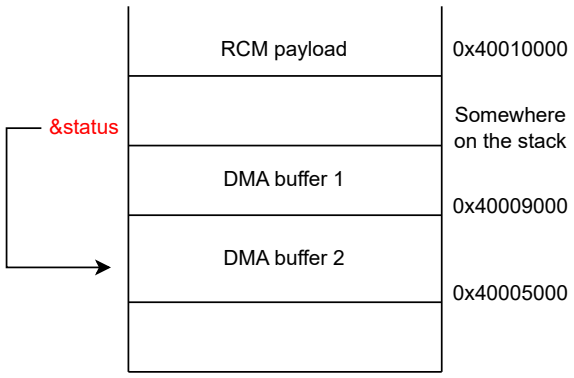

Figure 5.1: Fusée Gelée buffer overflow

Block	Type	Description
0x101474	avoid	Avoid checks for PMC scratch bit.
0x101328	avoid	Avoid entering bootloader launching.
0x101134	required	Required to enter the normal_boot function.
0x10142a	required	Required to enter the main function.
0x102896	required	Required to enter the function to load from recovery mode.
0x102896	target	Subsequent fuzzing should start execution from this address.

Table 5.1: Guidance configuration for Fuzzware

5.2 Fuzzing Setup

Configuring the emulation environment involves two main components: general setup and target-specific configuration. For the general setup, the correct memory layout and CPU architecture must be specified in the `config.yml` file. This information can typically be found in the device's datasheet. In the case of the BootROM firmware, it is executed by the booting CPU (ARM7TDMI) rather than the main CPU. Therefore, we need to use `UC_MODE_ARM926` instead of `UC_MODE_ARM` to set up the Unicorn instance and support the ARMv4T architecture. For Angr, specifying `ARM` as the simulated architecture is sufficient.

Beyond the general setup, target-specific configurations are required to guide the fuzzer and enhance emulation efficiency. Fuzzware provides options to specify which basic blocks must be executed and which should be avoided. Additionally, the fuzzer can be directed to the starting point of the logic that is of interest. When the target block is reached, Fuzzware records the current input as a prefix input, which is then used in subsequent fuzzing iterations. A detailed overview of these fuzzing guidance options is provided in Table 5.1.

Additionally, we implemented several function handlers to enhance fuzzing efficiency and facilitate the debugging process. To avoid providing excessive guidance to the fuzzer, we adhered to the following principles. First, we replaced functions that require numerous MMIO accesses

but contribute minimally to the program logic, such as `delay_us`. Second, we bypassed functions that perform certain checks, like the verification of USB connection setup. Third, we skipped unimportant functions that interfere with emulation, such as `debug_output`. Lastly, we replaced some standard library functions to aid in debugging. For example, functions like `memset` involve excessive block loops, complicating the analysis of basic block traces.

A key consideration was setting the USB control request buffer (a DMA address) as an MMIO address. This ensures that Fuzzware treats access to the request buffer as fuzzable. The format of the control request is detailed in Table 5.2. Since the USB control request pointer is located at address 0x40003978, we need the fuzzer to generate a sufficiently large 2-byte value at address 0x4000397e.

Field	Type
direction	1 bit
type	2 bits
recipient	5 bits
request	8 bits
value	16 bits
index	16 bits
length	16 bits

Table 5.2: USB request packet format.

5.3 Results and Discussion

After running the fuzzing campaign for 24 hours, we confirmed that the buffer overflow was successfully triggered. As shown in the crash output in Figure 5.2, the program counter (PC) value at the time of the crash is 0, indicating that the return address on the stack was overwritten. The length value provided by the fuzzer is 0xfbfb, which is sufficiently large to overwrite the entire stack. Notably, this vulnerability does not impose strict requirements on the input format.

Only two bytes in the USB request packet are needed to trigger this buffer overflow. However, Fuzzware may not be as effective when testing targets that require well-formatted packets for communication or in scenarios where reaching deeper logic involves building complex program states. This limitation arises because Fuzzware lacks an inherent understanding of input formats and often disrupts these formats during input mutation.

Another point worth mentioning is Fuzzware's underlying emulator, Unicorn. Unicorn is designed solely for instruction emulation, with all additional features, such as interrupt handling, requiring manual implementation. For more complex firmware, especially those utilizing the Linux kernel, Hoedur is a more suitable choice. Hoedur leverages QEMU as its base emulator, with its primary functionalities implemented in a loosely coupled manner with QEMU, allowing for a more comprehensive emulation of complex systems.

```
addr: 4000397f, rec_val: fb
addr: 4000397e, rec_val: fb
read from 0x4000fcb8 to 0x40005000, 0xfbfb bytes, till 0x40014bfb
0x4000fca8
        >>> [ 0x00000000 ] INVALID FETCH: addr= 0x0000000000000000
Execution failed with error code: 8 -> Invalid memory fetch (UC_ERR_FETCH_UNMAPPED)

==== UC Reg state ====
r0: 0x0000001a
r1: 0x00000001
r2: 0x40004000
r3: 0x00000000
r4: 0x00000000
r5: 0x00000000
r6: 0x00000000
r7: 0x00000000
r8: 0x00000000
r9: 0x00000000
r10: 0x00115be4
r11: 0x00000000
r12: 0x00000000
lr: 0x001073c3
pc: 0x00000000
cpsr: 0x200001d3
sp: 0x4000fce0
other_sp: 0x00000000

==== UC Stack state ====
0x4000fcd0: 00000000
0x4000fcd4: 00000000
0x4000fcd8: 00000000
0x4000fcdc: 00000000
0x4000fce0: 00000000 <---- sp
0x4000fce4: 00000000
0x4000fce8: 00000000
0x4000fcec: 00000000
0x4000fcf0: 00000000
0x4000fcf4: 00000000
0x4000fcf8: 00000000
0x4000fcfc: 00000000
0x4000fd00: 00000000
0x4000fd04: 00000000
0x4000fd08: 00000000
0x4000fd0c: 00000000
0x4000fd10: 00000000
0x4000fd14: 00000000
0x4000fd18: 00000000
0x4000fd1c: 00000000
======================


==== UC Other Stack state ====
======================

Emulation crashed with signal 11
```

Figure 5.2: Tegra X1 BootROM crashing output

Chapter 6

Related Work

In general, research on firmware fuzzing in emulated environments can be categorized into three main approaches. First, some studies focus on utilizing library abstractions. Several of these works [3, 12, 26] rely heavily on abstractions from the Linux kernel, making them unsuitable for monolithic firmware. Additionally, other studies [4, 15, 17] use Hardware Abstraction Layers (HALs). However, these approaches demand significant manual effort and specialized domain knowledge to identify and replace library functions.

An alternative line of research circumvents these limitations by forwarding peripheral access to the original hardware. While hardware-in-the-loop approaches [5, 10, 11, 14, 18, 23, 25] enable dynamic emulation of firmware, their slow forwarding speed reduces fuzzing efficiency. Moreover, the requirement for at least one hardware unit per fuzzing instance poses a challenge to scalability.

More recent efforts address MMIO access through pattern-based MMIO behavior modeling. Pretender [8] monitors hardware-firmware interactions and applies machine learning to recognize peripheral access patterns, although it requires access to the original hardware. P2IM [6] overcomes this limitation by dynamically matching peripheral accesses to predefined patterns within an emulated environment. μ-emu [27] employs guided symbolic execution to collect and resolve constraints related to peripheral registers. Fuzzware [20] and Hoedur [21] introduce coverage-guided fuzzing to model peripheral behaviors. Notably, all these works currently support only Cortex-M

Chapter 7

Conclusion

In this work, we investigated the scalability and usability of the cutting-edge rehosting system, Fuzzware. Focusing on rehosting monolithic firmware, we examined the differences between Cortex-M and Cortex-A (ARM32). We designed and implemented a Cortex-A extension for Fuzzware and validated this extension by exploiting the Fusée Gelée vulnerability in the Tegra X1 BootROM. Additionally, we enhanced Fuzzware's dynamic analysis capabilities by implementing a GDBServer and developed an experimental method to automatically synthesize handler functions to assist analysts. Our findings suggest that extending Cortex-A on Hoedur could be a more suitable approach for testing more complex firmware, particularly those involving the Linux kernel.

Bibliography

[1] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. California, USA. 2005, pp. 10–5555.

[2] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation". In: *Proceedings of the 36th Annual Computer Security Applications Conference*. 2020, pp. 746–759.

[3] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. "Towards automated dynamic analysis for linux-based embedded firmware." In: *NDSS*. Vol. 1. 2016, pp. 1–1.

[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)*. 2020, pp. 1201–1218.

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

[6] Bo Feng, Alejandro Mera, and Long Lu. "{P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1237–1254.

[7] Free Software Foundation. *GDB Remote Serial Protocol*. Accessed: 2024-07-10. GNU Project. 2023. URL: `https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html`.

[8] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. "Toward the analysis of embedded firmware through automated re-hosting". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019, pp. 135–150.

[9] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. "{PARTEMU}: Enabling Dynamic Analysis of {Real-World}{TrustZone} Software Using Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 789–806.

[10] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. “Embedded security testing with peripheral device caching and runtime program state approximation”. In: *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*. Vol. 118. 2016, p. 118.

[11] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. “Prospect: peripheral proxying supported embedded code testing”. In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 2014, pp. 329–340.

[12] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. “Firmae: Towards large-scale emulation of iot firmware for dynamic analysis”. In: *Proceedings of the 36th Annual Computer Security Applications Conference*. 2020, pp. 733–745.

[13] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

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

[15] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. “From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware”. In: *arXiv preprint arXiv:2107.12867* (2021).

[16] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. “Simics: A full system simulation platform”. In: *Computer* 35.2 (2002), pp. 50–58.

[17] Dominik Maier, Lukas Seidel, and Shinjo Park. “Basesafe: Baseband sanitized fuzzing through emulation”. In: *Proceedings of the 13th ACM conference on security and privacy in wireless and mobile networks*. 2020, pp. 122–132.

[18] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. “Avatar 2: A multi-target orchestration platform”. In: *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* Vol. 18. 2018, pp. 1–11.

[19] NGUYEN Anh Quynh and DANG Hoang Vu. “Unicorn: Next generation cpu emulator framework”. In: *BlackHat USA* 476 (2015).

[20] 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)*. 2022, pp. 1239–1256.

[21] Tobias Scharnowski, Simon Wörner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. “Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs.” In: (2023).

[22] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. "Firmfuzz: Automated iot firmware introspection and analysis". In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. 2019, pp. 15–21.

[23] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. "Charm: Facilitating dynamic analysis of device drivers of mobile systems". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 291–307.

[24] Fish Wang and Yan Shoshitaishvili. "Angr-the next generation of binary analysis". In: *2017 IEEE Cybersecurity Development (SecDev)*. IEEE. 2017, pp. 8–9.

[25] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." In: *NDSS*. Vol. 14. 2014. 2014, pp. 1–16.

[26] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. "{FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1099–1114.

[27] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. "Automatic firmware emulation through invalidity-guided knowledge inference". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2007–2024.