Forming Faster Firmware Fuzzers

Lukas Seidel[1], Dominik Maier[2], and Marius Muench[3]

[1] *Qwiet AI, jlseidel@qwiet.ai*
[2] *TU Berlin, dmaier@sect.tu-berlin.de*
[3] *VU Amsterdam and University of Birmingham, m.muench@bham.ac.uk*

Abstract

A recent trend for assessing the security of an embedded system's firmware is *rehosting*, the art of running the firmware in a virtualized environment, rather than on the original hardware platform. One significant use case for firmware rehosting is fuzzing to dynamically uncover security vulnerabilities.

However, state-of-the-art implementations suffer from high emulator-induced overhead, leading to less-than-optimal execution speeds. Instead of emulation, we propose near-native rehosting: running embedded firmware as a Linux userspace process on a high-performance system that shares the instruction set family with the targeted device. We implement this approach with SAFIREFUZZ, a throughput-optimized rehosting and fuzzing framework for ARM Cortex-M firmware. SAFIREFUZZ takes monolithic binary-only firmware images and uses high-level emulation (HLE) and dynamic binary rewriting to run them on far more powerful hardware with low overhead. By replicating experiments of HALucinator, the state-of-the-art HLE-based rehosting system for binary firmware, we show that SAFIREFUZZ can provide a *690x* throughput increase on average during 24-hour fuzzing campaigns while covering up to 30% more basic blocks.

1 Introduction

Embedded systems have become ubiquitous. These special-purpose computing devices are employed in all areas of everyday life, such as automotive systems, networking equipment, healthcare machines, smart-home devices, and more. Since they become evermore connected, protecting their confidentiality, integrity and availability gets increasingly important. In contrast to traditional computers, embedded devices often do not run a full-fledged operating system, but a monolithic software stack that handles every aspect of the system: memory management, interrupts, processing of user data, and hardware interactions. This so-called device firmware often exposes a lot of complex functionality, such as drivers and custom parsers, to external sources that may be attacker-controlled, for example via wireless data over the air, or even from TCP packets. Given the importance of embedded systems and the extended attack surface, testing their firmware's security is essential.

A common approach for security analysis is fuzzing, an automated process in which a fuzzer feeds semi-random, potentially malformed input to the target, to find buggy corner cases. Due to its effectiveness, the field has become an increasingly popular research topic [41]. The recent field of *rehosting* commonly enables fuzzing of an embedded device's firmware by creating virtual execution environments [24, 47]. To automatically generate inputs that are accepted by the target, modern fuzzers use feedback about the target's last execution to choose which inputs to mutate further. As the inputs gradually cover a larger amount of the target's code for each execution, throughput naturally decreases. Since every testcase spends more time in the target, the fuzzer can execute fewer testcases per second, which slows down further exploration of the state space. Hence, one angle to further increase fuzzing efficacy is to vastly increase execution speed. Testing the program against the same inputs in less time gives the fuzzer more time to explore the target further.

In this paper, we propose SAFIREFUZZ, a new and performant rehosting and fuzzing approach for embedded binary-only ARM firmware. Instead of rehosting target firmware using a general-purpose emulator, as done by prior work (e.g., [20,33,35,46,50]), we deploy a technique which we term *near-native* rehosting. The core insight behind our approach is that powerful server and desktop ARM computing devices provide execution modes and instruction sets sufficiently similar to the ones present on embedded systems. Based on this observation, we create a dynamic binary rewriting engine to run firmware directly on a more powerful system, while inserting fuzzing instrumentation on-the-fly. To deal with hardware interactions, we follow the High-Level Emulation (HLE) approach proposed by HALucinator [20] and replace hardware interactions with HAL-based hooks.

As we will show, our approach significantly improves execution speed when compared with recent rehosting frameworks, leading to improved fuzzing efficacy and the discovery of previously undetected bugs. In particular, we evaluate SafireFuzz against HALucinator [20], the state-of-the-art HLE rehosting approach, and Fuzzware, a recent peripheral-modeling-based rehosting approach. Our evaluation shows that SafireFuzz can provide an up to 690x speedup compared to HALucinator and up to 147x compared to Fuzzware, resulting in the additional discovery of up to 30% additional basic blocks during 24-hour fuzzing runs.

In summary, we make the following contributions:

- We propose SafireFuzz: a high-performance near-native rehosting framework for interactive execution of embedded ARM firmware.
- We prove its applicability for highly efficient fuzzing of ARMv7-M binary firmware images. For this, we tightly integrate in-process fuzzing with dynamic binary rewriting techniques, in conjunction with Hardware Abstraction Layer (HAL) function hooking.
- We evaluate SafireFuzz by implementing fuzzing harnesses for 12 firmware samples from the HALucinator test suite and compare its performance against recent rehosting approaches. We also rehost two new samples from scratch. Our evaluation shows that near-native rehosting outperforms rehosting approaches built on top of general-purpose emulators.

2 Background

2.1 Embedded Systems & Firmware

Embedded systems are characterized by their use of firmware, which is responsible for driving the device's hardware and offering higher-level functionalities at the same time. For interacting with the hardware, the firmware typically uses one of the following channels.

Memory-Mapped Input/Output (MMIO) assigns a range in the physical memory to each peripheral. Each of these ranges is divided into MMIO registers. Accessing these registers allows the firmware to directly interact with its peripherals, for instance, to read data from an external source or to turn on a LED. *Port-Mapped Input/Output* (PMIO) behaves similarly to MMIO, with the difference that specialized instructions enable the interaction via IO ports. *Direct Memory Access* (DMA) allows firmware to bypass the CPU when transferring data between peripherals and main memory. Instead of blocking the CPU for the whole duration of the slow memory transfer, the CPU only needs to initiate the process by interacting with a specialized DMA controller peripheral. Peripherals indicate the occurrence of specific events (e.g., the arrival of new data) via *Interrupts*. Based on the configuration of the device's interrupt controller, the firmware then resumes execution at a designated *Interrupt Service Routine* (ISR).

2.2 ARM Cortex-A/M

ARM is one of the most popular instruction set architecture families for embedded systems [12]. In particular, the 32-bit ARMv7-M and ARMv7-A variants are widely used, due to their low cost and energy efficiency. ARMv7-A, targeting more complex embedded systems, features two different execution modes: In ARM mode, the processor executes instructions with a fixed size of four bytes at a four-byte alignment. In Thumb mode, instructions consist either of two or four bytes, and the resulting two-byte alignment allows for much denser packing of code [3] which is favorable for resource-constrained embedded systems. These modes are switchable on the fly via the—otherwise unused—least significant bit for branch targets, whereas a `1` indicates execution in thumb-, and `0` execution in ARM mode at the target location. ARMv7-M, on the other hand, specifically targets microcontrollers and only implements the Thumb-v2 instruction set.

Beyond that, the more recent instruction set families ARMv8-A and ARMv9-A added the AArch64 extension, which provides a 64-bit instruction set. While CPUs implementing these families usually target mobile and desktop devices, the extension provides support for executing in 32-bit mode on lower exception levels (i.e., EL1 and EL2) while using a 64-bit OS or hypervisor.

2.3 Fuzzing

Fuzzing is a popular approach for automatic vulnerability discovery, able to uncover a multitude of different vulnerabilities. These include, but are not limited to, memory corruption bugs, such as buffer overflows, double frees and use-after-frees, and logic bugs, such as integer overflows, infinite loops and even race conditions. In coverage-guided fuzzing, the fuzzer uses execution feedback to determine interesting inputs. For this, the fuzzer adds instrumentation to the target program, either at compile-time, if source code is available, or later on a binary level. This instrumentation reports coverage information back to the fuzzing engine, e.g., by tracking executed branches in a bit map. When an input generates unique coverage, it is added to the fuzzing corpus to be subsequently mutated for the generation of new test cases.

3 Motivation

Rehosting, the automated creation of virtual execution environments of embedded firmware, combines multiple approaches to overcome the challenges associated with emulating the (potentially unknown) peripherals of an embedded system [24]. While early work relied on hardware-in-the-loop emulation to offload unknown accesses to a physical device [35, 37, 48], hardware-less rehosting became the de-facto standard to enable coverage-guided fuzzing of embedded systems [20, 25, 31, 33, 40, 45, 46, 51].

Rehosting System	Approach	Emulator	Binary
PRETENDER [31]	Pattern-based	QEMU	y
HALucinator [20]	HAL-based	Unicorn	y
PartEmu [32]	Pattern-based	QEMU	y
P2IM [25]	Pattern-based	QEMU	y
Frankenstein [45]	HAL-based	QEMU	y
Para-Rehosting [38]	HAL-based	N/A	n
JetSet [34]	Symbolic Execution	QEMU	y
uEmu [51]	Symbolic Execution	S2E	y
FirmWire [33]	Pattern-based	Panda	y
FUZZWARE [46]	Symbolic Execution	Unicorn	y
SEmu [52]	Specification-guided	QEMU	y*

* Requires access to device documentation.

Table 1: Survey of recent firmware rehosting solutions.

Scharnowski et al. [46] classify the approaches to overcome unknown peripheral behavior deployed by hardware-less rehosting approaches in three categories: (1) high-level emulation, (2) pattern-based MMIO modeling, and (3) symbolic-execution-based approaches. The first category aims to eliminate hardware accesses of virtualized firmware by hooking library functions of the hardware-abstraction layer. The second approach uses heuristics to categorize MMIO registers and then uses pre-defined models to respond to accesses. The third approach aims to resolve values advancing firmware execution on-the-fly via symbolic execution. Additionally, recent work [52] proposes specification-guided emulation in which MMIO peripheral models are derived from datasheets and device documentation.

We survey recent rehosting systems in Table 1 and note that all approaches have been used to implement fuzzing campaigns for firmware. However, we also observe that, except for Para-Rehosting [38] which requires source code access, all surveyed rehosting systems rely on already existing emulators to virtualize the firmware. We hypothesize that this is a hindrance to fuzzing, as readily available emulators are general-purpose tools and were never designed with fuzzing in mind.

A pathway to faster firmware fuzzing. Closer inspection of Table 1 yields another detail: All surveyed emulation-based rehosting systems either extend QEMU directly or build on top of a QEMU-based emulator. This raises QEMU's emulation approach to the de-facto standard for firmware rehosting. While QEMU offers huge extensibility and support for various ISAs, we believe that relying solely on its emulation capabilities leads to trade-offs for fuzzing efficacy. In particular, when inspecting the state of the art, we observe the following commonly accepted performance roadblocks:

[R1] Binary Lifting & Recompilation. QEMU lifts guest code to TinyCode, its internal intermediate representation, before applying instrumentation and Just-In-Time (JIT) compiles each block to the host architecture. While this results in support for various instruction sets, most rehosting work focuses solely on the ARM architecture. Hence, we argue that a high-throughput firmware fuzzer may consider alternative strategies for additional performance, such as direct binary translation or binary rewriting.

[R2] Expensive Dispatch of Memory Accesses. Monolithic firmware usually resides in a single flat address space. However, QEMU was developed for more complex systems deploying an MMU. To emulate this across all systems, its so-called *SoftMMU* dispatches memory accesses. This leads to significant performance overhead [17]. Being able to directly access the guest memory without indirection would greatly benefit guest execution speed, and hence, fuzzing.

[R3] Basic Block Caching & Chaining. One of the core performance optimizations of QEMU is the ability to cache already translated blocks and chain the execution of multiple blocks together. However, this optimization was not available in early adaptations of AFL-QEMU [11]. While mainlined in AFL++ [26] and resulting, we observe that various rehosting solutions were developed on top of legacy versions and the resulting lack of adoption severely hinders fuzzing performance.

[R4] Lack of In-Process Fuzzing. Up to now, rehosting solutions run their fuzzing engine in a separate process. However, this leads to unnecessary kernel interaction and context switches compared to a solution that embeds the fuzzer in the same process.

We note that some roadblocks were partially addressed by prior work. For instance, FirmWire [33] deploys the basic block caching & chaining optimizations and Frankstein [45] uses QEMU's user mode which eliminates the need for a SoftMMU. However, to the best of our knowledge, no prior work systematically tackled all roadblocks and explored the possibility of a highly performant firmware fuzzer.

4 Design

4.1 Overview

SafireFuzz acts as a highly-efficient rehosting and execution engine for firmware fuzzing by overcoming the roadblocks described in Section 3. At the core of our proposed approach stands a technique we term *near-native rehosting*. Instead of emulating the firmware through lifting and recompilation, *(R1)*, we exploit the fact that certain ARMv8-A cores provide userspace compatibility with the AArch32 and Thumb instruction set variants. Hence, we can directly execute large parts of the firmware code on powerful cores through binary instrumentation. As we mirror the memory layout of the embedded device in userspace, rewritten instructions do not need additional logic to dispatch memory accesses, circumventing *(R2)*. Additionally, our rewriting approach is optimized to cache already instrumented blocks, minimizing engine overhead *(R3)*. Lastly, we embed the fuzzing logic in the same process space as the engine and the rewritten firmware to minimize the required interactions with the host operating system *(R4)*.

In the following, we will describe SafireFuzz's core engine, our dynamic rewriting approach, as well as our solution to rehosting challenges.

4.2 Rehosting & Rewriting Engine

SafireFuzz's engine is responsible for executing the target firmware, handling rehosting aspects, rewriting instructions, and insertion of fuzzing instrumentation. For dealing with unknown hardware peripherals, we loosely follow the High-Level Emulation approach by Clements et al. [20], as hooking corresponding HAL embeds easily into our rewriting approach. Hence, the engine uses a firmware-specific harness. This harness initializes memory ranges and registers HAL hooks before target execution is started by rewriting the first basic block at the specified entry point.

The engine is responsible for translating basic blocks on demand while adding instrumentation for fuzzing, transferring execution to registered hooks where required, and deploying interrupt approximation mechanisms. While all of these tasks are crucial, we note that as little time as possible should be spent inside the engine's code or in the operating system. Hence, the engine only rewrites basic blocks when their immediate predecessor gets executed for the first time, and keeps a cache of already instrumented blocks. During the initial rewriting and execution of a block, multiple jumps to the engine may be required. After an emitted basic block has been executed completely for the first time, the engine eliminates all, now unnecessary, jumps from this block back to itself. This way, rewrites of successive blocks and branch resolution have to happen only once. The instrumentation overhead is constant for one engine run.

4.3 Basic Block Rewriting

The simplest approach to dynamic rewriting is the in-place replacement of instructions. While solutions following this approach exist to replace single instructions [22], we argue that this is not possible in the general case, and especially not for ARMv7-M, as replaced instructions may be larger than the original ones. On top, we need to insert additional instructions for instrumentation and hooking. This shifts the relative position of instructions to each other. As many instructions on ARM operate in a PC-relative fashion and jump targets cannot be guaranteed to be preserved. While calculating targets of dynamic branches and the required alterations are possible with control-flow recovery heuristics [44], inferring the register contents statically is non-trivial. Consequently, we instrument the binary dynamically at runtime. As any basic block is never rewritten more than once, the approach's one-time overhead is negligible. We describe the rewriting process in Algorithm 1 and further illustrate the rewriting process on a basic block level in Figure 1. One upside of our near-native rehosting approach is that a majority of all instructions require no rewriting at all, as there is no mismatch between the AArch32 execution state on the Cortex-A core we are utilizing and the Cortex-M the firmware was built for. Cortex-M processors make use of Thumb-v2 instructions, of which most can be executed natively and without divergence on an ARMv8-A target platform with AArch32 mode support[1]. PC-modifying instructions and PC-relative memory accesses are the two only classes of instructions that require rewriting. This stands in contrast to the usually deployed processor-emulation-based rehosting techniques, where instructions for one architecture are executed on another and all instructions require translation.

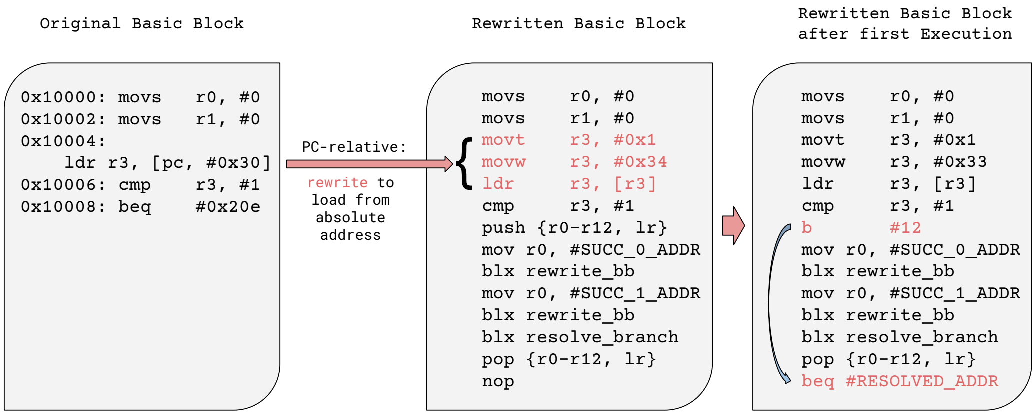
Algorithm 1 Binary Rewriting in SafireFuzz.

```
for curr_addr in basic_block do
    if curr_addr in registered_hooks then
        insert(jump_to_hook)
        break
    end if
    insn = disas(curr_addr)
    if requires_rewriting(insn) then
        insert(patch(insn))
    else
        insert(insn)
    end if
    if is_delimiter(insn) then
        insert(branch_to(rewrite_successors))
        insert(branch_to(resolve_branch_target))
        break
    end if
end for
copy(*execution_code_site, rewritten_basic_block)
```

4.4 Function Hooking

While rewriting a new basic block, the engine checks whether a hook is registered for the current address. A user can supply such a function, written in a high-level programming language. The engine emits a jump to the user-supplied code that will then execute at block execution time. The hooking locations in SafireFuzz are restricted to function hooks by design. As all (register) state we alter is shared between

[1] Exceptions are low-level system instructions like `svc`, `swi`, or `mrc`/`mcr`. However, due to our HAL-based rehosting approach, functions including them are never executed.


Original Basic Block
Rewritten Basic Block
Rewritten Basic Block after first Execution
PC-relative:
rewrite to load from absolute address

```
0x10000: movs   r0, #0
0x10002: movs   r1, #0
0x10004:
    ldr r3, [pc, #0x30]
0x10006: cmp    r3, #1
0x10008: beq    #0x20e
```

```
movs    r0, #0
movs    r1, #0
movt    r3, #0x1
movw    r3, #0x34
ldr     r3, [r3]
cmp     r3, #1
push {r0-r12, lr}
mov r0, #SUCC_0_ADDR
blx rewrite_bb
mov r0, #SUCC_1_ADDR
blx rewrite_bb
blx resolve_branch
pop {r0-r12, lr}
nop
```

```
movs    r0, #0
movs    r1, #0
movt    r3, #0x1
movw    r3, #0x33
ldr     r3, [r3]
cmp     r3, #1
b       #12
mov r0, #SUCC_0_ADDR
blx rewrite_bb
mov r0, #SUCC_1_ADDR
blx rewrite_bb
blx resolve_branch
pop {r0-r12, lr}
beq #RESOLVED_ADDR
```

Figure 1: Example of SafireFuzz rewriting a basic block. The new block resides somewhere in the rewritten code site (cf. Fig. 2). Jumps into the engine are simplified and require setting up the branch target registers in reality. After the first slow-path execution, jumps back into the engine are skipped on subsequent hot-path executions. Context save and restore are also simplified and usually involve the preservation of processor condition flags.

the executed firmware and user code, writing hooks on a per-instruction base would be more complex when harnessing a new target. Also, we found that function hooks are in all cases sufficient to stub out hardware interactions for HLE rehosting. This restriction allows us to gain runtime performance, as we can make certain assumptions in our state save and restore routines, as long as the firmware respects common ARM calling conventions. Our approach's introduced overhead is minimal: The context save and restore, including the jump into the hook, comprises five instructions. Such hooks, also called *handlers*, are designed to model the behavior of a part of the firmware. They replace calls to functions on the Hardware Abstraction Layer during execution and thereby mask peripheral accesses in the firmware. This procedure is the core of peripheral management in HLE-based systems and our near-native approach introduces no intrinsic drawbacks. Commonly implemented functionalities are simulation and handling of interrupts, accepting external data during fuzzing and making it available to the firmware, or replacing the memory allocator with a sanitizing one to increase the observability of security violations. As we model functions on the HAL in our approach, cross-firmware reuse is facilitated. Firmware images using the same system libraries or being developed for the same microcontroller often share common hardware abstractions. Which firmware functions are hooked with which handler is determined by the user within a *harness* (cf. 5.3). A firmware's harness can be seen as a specification of domain knowledge in code. They are highly specific to a certain processor or microcontroller model and have to take care of handling a firmware's characteristics, from setting an entry point to providing correctly mapped memory.

4.5 Interrupt Approximation

Many embedded devices rely on interrupts for signal delivery and the processing of asynchronous, external events. The *SysTick* timer, for instance, is present in many ARM microcontrollers. Monolithic Real-Time OSs (RTOS) use it to poll MMIO registers and schedule new tasks. Therefore, our rehosting solution requires a way of simulating interrupts in order to accurately execute such firmware.

For this purpose, we implement tick-based interrupts. Although it would be theoretically possible to translate interrupts and let the Cortex-A host system handle them in our near-native scenario, interrupts on embedded devices are commonly triggered by external peripherals and their ISRs will try to access them via MMIO, hence requiring a rewrite in any case. Previous rehosting work has shown that interrupt approximation is sufficient to model firmware behavior [25] and we see it as beneficial for fuzzing: our tick-based counter leads to higher determinism, thus resulting in reproducible and analyzable program traces. Improving on HALucinator, which uses a similar approach with basic-block-level counters to trigger a timer, our approach uses indirect call-level counters and manual clock-update hooks. We update timers and trigger interrupts at specific points in the execution. Hooking every single basic block would introduce unnecessary performance overhead.

5 Implementation

5.1 Engine Internals

We implemented SafireFuzz using the *Rust* programming language. The engine core consists of 1481 source lines of code. Another 1716 make up the entirety of implemented

HAL handlers, the harnesses for the 14 evaluated targets add up to 2360 lines. Disassembling is performed with *Capstone* [4]. For the assembly of modified or new instructions to be emitted, we use *Keystone* [7].

The engine handles all tasks to allow the execution of firmware in a foreign environment. Hence, it performs many tasks an emulator has to take care of as well.

Branch Resolution. When the engine rewrites a basic block that closes with a static branch, it emits a call back into the engine. On the first execution of the basic block, this jump into the engine, also called *slow path*, gets replaced with a static branch, resolving the original firmware address to the new location of the rewritten basic block. As such branch target calculations only need to be performed once, the removal of the jump into the engine reduces the overhead on forthcoming executions. Executing all these slow path functions exactly once per basic block and skipping them afterward is a major performance optimization.

If instructions directly modify the PC in a not statically-resolvable way, i.e., using register contents, the engine needs to resolve the target at runtime. These include `BX`, `BLX` and `MOV` instructions with PC as the target register. The routine resolving the correct address of the target basic block in the rewritten code range performs the following steps: If the target address resides in our rewritten code range, we assume a tailcall. We then simply *OR* the address with 1 to make sure that we stay in Thumb mode and return this address. Otherwise, a cache lookup is performed. The engine rewrites the new basic block if it is not already cached. Subsequently, the real address of the rewritten block is returned.

The branch resolution function is also one of the anchors for our engine's interrupt approximation. Interrupts can be registered as execute-every-nth-tick, where a tick is a BLX jump or the execution of a manually placed hook. Every time we resolve a dynamic jump, a branch counter is increased. If interrupts are globally enabled, and any interrupt handlers are registered, the engine triggers the interrupt if enough ticks have passed.

Jump tables are commonly compiled as loads of a fixed value from the data segment directly into the PC. When the engine encounters such `LDR PC, [...]` instructions upon rewriting, it emits a jump back into the engine. This handler function checks whether the specific load was already resolved and cached.In the case of the first invocation, the new basic block gets rewritten, and the memory location the load reads from is overwritten with the new address. On return from the engine, the original instruction is executed and reads the adjusted value, correctly adjusting the PC. ARMv7-M additionally features Table Branches, causing a PC-relative branch using an offset table. Again, execution is redirected back to the engine where it performs the table lookup and calculates the corresponding offset. With this information, the target basic block is instrumented and lifted, if it is not already cached. Finally, the address of the branch target is returned in $r0$. In the new basic block, we overwrite the register-specific offset on the stack with the returned value, giving us control over the register content after restoring the execution context. We provide additional details on how we handle further control-flow modifying instructions in Appendix A.2.

Context Switching. All jumps back into the engine require a context save and restore. In case of a jump into user-defined hooks, the routine is minimal: The engine pushes $r1$ to $r11$ on the stack and pops them again upon the hook finishing. Finally, the engine branches back to the Link Register (LR). This approach only induces negligible overhead. Furthermore, it allows access to function parameters and the returning of values in a more natural way than in emulators such as Unicorn. While in Unicorn a hook has to use API calls to read and write register contents, our framework exposes the natural Application Binary Interface (ABI). Before resolving branch targets in the engine, a full context save is required. This includes all 13 general-purpose registers, the LR, as well as condition flags in the Application Program Status Register (APSR).

Memory Accesses. After branches, PC-relative memory accesses are the second-largest class of instructions the engine needs to modify when rewriting basic blocks to another location. In ARMv7-M, small chunks of data are often co-located with the basic blocks that load from them, using a PC-relative `LDR`. As it is non-trivial to infer how large the data segment behind a basic block will be, copying the data during basic block rewriting is non-trivial. We therefore statically resolve the address and replace the PC-relative load with an absolute one from the original code site. Figure 2 shows the engine's memory layout.

Caching. To minimize time spent inside the engine during hot-path execution, we cache various data points.
Reassembling instructions with Capstone and Keystone is very expensive. Consequently, we cache and reuse blocks wherever possible. One example are wide branches (`B.W`) emitted when replacing branches referring to locations in the original code site with branches targeting rewritten blocks. As the machine code instruction encodes an offset and not an absolute address, it has a high potential for reuse. We insert the assembled bytes into a fixed-size array and perform the lookup by the required offset. Dynamic branching instructions such as BLX have to get resolved every time because register contents and hence the jump target might change. This requires a lookup, using the original address in the firmware to retrieve the equivalent basic block's location in the new code site. As our map *key* in both cases is a value in a linear address space with a well-defined upper bound, we can use a simple array as

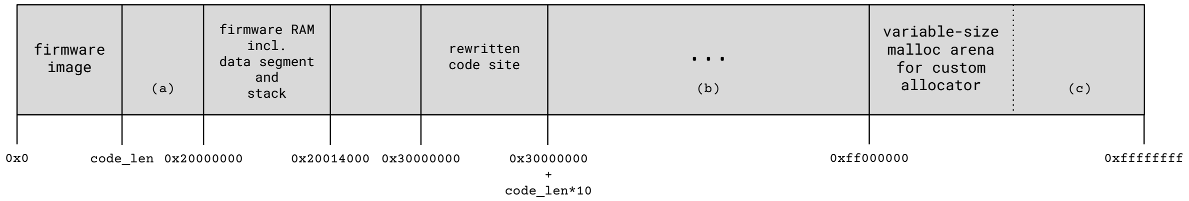


Figure 2: Example Memory Layout of SAFIREFUZZ highlighting rehosting-specific regions. The rewritten code site contains the new basic blocks after processing through the engine. Firmware execution happens here. Regions in parentheses include usual process data: (a) contains the normal code segment, (b) contains shared libraries, (c) contains the program's stack.

the data structure. Here, the *i-th* element contains the address of the rewritten basic block for the original address *i*. This allows us to do lookups in one instruction. We store such a mapping not only for targets of BLX instructions but for every address for which the engine emitted a new block.

Processor Cache Maintenance. As our engine not only emits new instructions once but also modifies already-emitted basic blocks, we have to deal with the ARMv7-A core's non-unified cache architecture. Such cores have separate, non-coherent caches for instruction and data accesses, potentially leading to problems and heavy inconsistencies when it comes to self-modifying code. While it is possible to overwrite instructions in memory at runtime, the processor might still execute old or invalid instructions due to missing coherence. After every instruction rewrite, i.e., overwriting an instruction in memory that was already executed at least once, we need to invalidate both caches for the corresponding memory range. The next fetch on these instructions will then cache-miss and the processor will correctly load the new version from memory. Cache flushes as well as cache misses are the exception. They are strictly necessary to guarantee the coherence of our rewriting approach, and the observed overhead during testing was negligible.

5.2 The Fuzzer

A core building block of our implementation is the fuzzer. LibAFL [27] provides the mutation backend. We chose its on-disk corpus to store the input queue as well as found objectives. The harness acts as the entry point to our engine and defines the firmware from the fuzzer's point of view. First, it retrieves the fuzzing input from LibAFL and then kicks off a single execution. As feedback mechanisms, we use a combination of a map observer, tracking the state of the coverage map, which is updated on every conditional branch, as well as the execution time and timeouts. For scheduling, we employ a strategy favoring small test cases. Mutations are performed following AFL's *havoc* approach, as implemented in LibAFL. The method involves bit flips, integer overwrites, block deletion and block duplication. Our framework also exposes the option to specify a token file the fuzzer will use to mutate new inputs. Tokens in AFL are domain-specific byte sequences, such as tags in HTML or XML, facilitating the fuzzer's job to generate meaningful input. LibAFL's `Launcher` component combines all these pieces and handles the launching and restarting of fuzzing processes.

Coverage Tracking. In order to enable coverage-guided fuzzing, the engine needs to track coverage and make it observable by the fuzzing backend. We implement non-colliding edge coverage tracking by setting values in a *static* bit map whose address and size are known at compile time. To track coverage, the engine inserts an additional basic block after every conditional and table branch. For every previously unseen edge, SAFIREFUZZ increases a global counter, acting as a unique identifier and index into the global bit map. The inserted instructions update the corresponding entry upon execution. This approach, with the new basic block consisting of seven instructions including only a single memory access, minimizes the introduced overhead while still providing the fuzzer with meaningful insights. We opted for a boolean coverage map as opposed to a *hitcounts* approach to further reduce overhead, as experiments have shown that plain edge coverage in many cases even outperforms AFL's default hitcount metric [28].

Parallelism. The LibAFL Launcher allows scaling to an arbitrary number of cores. Multiple fuzzing instances can be started automatically in parallel and information such as the coverage map is synchronized. Each core runs its own process with an individual engine instance without shared caches. This way we avoid executing expired or inconsistent views of the rewritten code, potentially resulting from one instance executing a basic block that is currently being rewritten by another, without expensive cache coherence attestation.

5.3 Harnessing

Harnesses in SAFIREFUZZ are supplied at compile time and are written in Rust, as is the rest of the framework. To allow

Firmware	HAL	# Hooked Functions
WYCINWYC	STM32	25
NXP HTTP	mcuxpresso	23
SAMR21 HTTP	SAMR21	23
6LoWPAN Receiver	Contiki	37
6LoWPAN Transmitter	Contiki	29
P2IM Drone	STM32	32
P2IM PLC	STM32	32
STM PLC	STM32	35
TCP Echo Client	STM32	31
TCP Echo Server	STM32	29
UDP Echo Client	STM32	31
UDP Echo Server	STM32	28

Table 2: Targets with their corresponding Hardware-Abstraction Layer and the amount of hooked functions necessary for successful rehosting.

flexible configuration while maintaining usability, the engine exposes a set of interfaces a harness' developer has to implement, i.e., functions the engine can call to handle various parts of the rehosting process. This includes but is not limited to a `setup` function, called once at engine initialization, e.g., to set up memory segments and copy the firmware image to the correct position, and a `reset` function handling memory restoration and resetting timers and the custom allocator.

6 Evaluation

In our evaluation, we set out to answer the following three research questions:

RQ1. How does SAFIREFUZZ compare to the state of the art in firmware fuzzing?

RQ2. What are the core performance gains and remaining roadblocks for SAFIREFUZZ?

RQ3. Can SAFIREFUZZ identify previously unknown or undetected vulnerabilities?

To answer *(RQ1)*, we select 12 firmware targets from previous research on firmware security and rehosting [20,25,30,31,42,51] and compare the SAFIREFUZZ's fuzzing efficacy with *HALucinator* [6] and *Fuzzware* [46] in different configurations. Based on these results, we provide a detailed analysis of SAFIREFUZZ's performance *(RQ2)*. Lastly, to answer *(RQ3)*, we first discuss previously undetected bugs in the 12 firmware samples found by SAFIREFUZZ and then apply our approach to two new targets.

Unless otherwise specified, we executed all experiments on a HoneyComb LX2 ARM workstation running a 64-bit Ubuntu 18.04. This system features 16 ARM Cortex-A72 cores with a clock rate of up to 2 GHz, 32 GB DDR4 memory with a frequency of 3200 MT/s and a 128 GB m.2 SSD.

6.1 Experiment Setup

Target Selection. We selected targets based on their prevalence in prior research with a special focus on HALucinator [20], as this is the most recent binary HAL-based rehosting framework. All targets chosen for evaluation are compiled for Cortex-M cores. The specific SAFIREFUZZ harnesses are following the hooking and implementation of HALucinator in order to ensure semantically identical behavior when being presented with the same input. We provide an overview of the targets in Table 2 and detail hooked functions on the Hardware Abstraction Layer for four exemplary targets in Appendix A.3.

Fuzzer Setup. We compare SAFIREFUZZ against HALucinator [20] and *Fuzzware* [46], the respective state-of-the-art HLE-based and peripheral-modeling based fuzzer.

To compare with HALucinator, we use *hal-fuzz*, its fuzzing-oriented open source version [6]. This version is based on UnicornAFL [10] and uses legacy AFL [1] as fuzzing backend. To investigate the impact of a more modern fuzzer and increase comparability with SAFIREFUZZ, we swap in a libAFL-based backend as the mutation engine. We apply the same configuration parameters and mutation strategies as the ones used by SAFIREFUZZ and term this setup *HALucinator - libAFL*. For Fuzzware, we use its AFL++ [26] fuzzing backend with AFL++v3.14c. Unfortunately, we encountered non-trivial bugs when running Fuzzware on our ARM platform which severely limited fuzzing performance. After consulting with the Fuzzware authors, which confirmed that ARM hosts are not supported, we resorted to running Fuzzware on an x86 host. We used a Ubuntu 18.04 VM with 64 cores and 196 GB of RAM, hosted on an AMD EPYC 7662 server.

For each fuzzer and target, we conducted five 24-hour runs with each fuzzing process pinned to one designated core.

Seed Selection. We use the seeds provided by HALucinator for all setups except for Fuzzware, where we use empty seeds. This is due to the different input semantics for the fuzzers: In contrast to HAL-level abstractions, it is not possible to use known file formats as seed inputs for Fuzzware, as its inputs encode the interactions with the different MMIO registers.

Fidelity. Execution flow on the basic block level cannot be guaranteed to be 100% identical for the different frameworks due to their implementation differences. We ensured that the simulated systems exposed functionally equivalent behavior between HALucinator and SAFIREFUZZ by comparing message logs and exit addresses.[2] All sample inputs provided

[2] We enabled debug prints at various places for all firmware targets, logging printable output to STDOUT and all interaction with HAL-I/O, e.g., packet contents when a simulated ethernet packet arrives. The produced logs provide a detailed trace of the input processing of the firmware under test.

Firmware	SafireFuzz		HALucinator		HALucinator - libAFL		Fuzzware	
	exec/s	# basic blocks	exec/s	# basic blocks	exec/s	# basic blocks	exec/s	# basic blocks
6LoWPAN Receiver	581.4	2840	1.2	2354	2.5	2724	73.6	1812 / 1618
6LoWPAN Transmitter	1877.0	2563	1.8	2176	2.6	2307	66.4	2460 / 2101
NXP HTTP	5216.8	2341	4.8	1990	4.5	2209	22.5	447 / 337
SAMR21 HTTP	2894.6	1927	3.1	1581	1.7	1310	1018.4	52 / 26
P2IM PLC	772.1	202	19.5	228	6.3	249	24.5	87 / 70
P2IM Drone	7279.7	237	9.3	281	2.8	283	9.7	583 / 500
STM PLC	7193.8	748	10.8	654	2.0	776	15.5	732 / 381
WYCINWYC	3083.1	3263	9.4	1384	12.3	2795	41.0	3375 / 3166
TCP Echo Client	3401.3	2403	4.8	1679	4.0	2290	87.2	460 / 375
TCP Echo Server	2762.1	2177	5.0	1563	4.7	1710	88.4	459 / 229
UDP Echo Client	4485.3	1613	5.0	1188	4.7	1594	90.2	460 / 229
UDP Echo Server	4636.7	1450	5.9	1045	5.1	1485	85.1	460 / 229

Table 3: Results of fuzzing the targets over 24 hours. Reported numbers are median values from the five runs. For Fuzzware, we report reached basic blocks both with and without considering HAL functions.

by the hal-fuzz repository were executed in both HALucinator and SafireFuzz and we asserted that the resulting logs matched. Additionally, we generated such message logs for various inputs from the fuzzing queue, selected at random, as well as for crashing inputs and made sure that the traces and exit addresses were equivalent.

Metrics. The metrics we use for comparative means are *(1)* executions per second and *(2)* total coverage measured in basic blocks.[3] For *(1)* we count the total executions for each fuzzing run and divide them by the 24-hour time budget. For *(2)*, we replay the test cases in the respective tools, except for SafireFuzz, where we replay found test cases in hal-fuzz for better comparability. As both Fuzzware and hal-fuzz are based on Unicorn, this allows us to collect *translated blocks*, which we further filter for actual basic blocks as defined by Ghidra's [5] *SimpleBlockModel*.

For Fuzzware, we replay the testcases a second time, while ignoring subtraces traversing HAL functions hooked by the other frameworks. This allows us to identify the number of basic blocks not executed by our HLE-based approach.

6.2 Comparison with the State of the Art

Table 3 shows an overview of the results of our experiments and Figure 3 visualizes reached coverage over time. In the following, we discuss SafireFuzz's performance in terms of execution speed and reached basic blocks in comparison to HALucinator and Fuzzware on our experimental platform. For additional analysis of found crashes and a comparison to other rehosting tools, we refer to Appendix A.1.

SafireFuzz vs. HALucinator. On all targets except the P2IM PLC firmware, our framework offers greatly increased performance compared to HALucinator. Conducting the Mann-Whitney U test on the execution speed and coverage metrics confirmed statistically significant divergence for $p < 0.05$ between SafireFuzz and HALucinator for all targets, with the exception of coverage for the P2IM PLC target. We note that the fuzzing campaign against this target is inefficient for all frameworks: the target exhibits extremely easily triggerable crashes and, additionally, a significant part of all inputs lead to an infinite loop and, thus, a timeout.

We achieve an up to 1000x increase in raw throughput when running the frameworks in the same environment. When considering reached basic blocks over time, we observe that fuzzing with HALucinator offers higher consistency and more reliable results. However, even in a worst-case comparison our approach is able to offer improvements for most targets.

SafireFuzz vs HALucinator-libAFL. When replacing legacy AFL with LibAFL in HALucinator, achieved coverage is greatly improved but still bested by our framework in nearly all cases during 24-hour runs. Performing the Mann-Whitney U test indicates statistical significance except for the coverage for ST PLC, WYCINWYC and the UDP Echo Client samples. In all of these cases, HALucinator-libAFL's overall achieved coverage is close, or better, than SafireFuzz's. As expected, the differences in execution speed of HALucinator-libAFL remain approximately the same compared to HALucinator. Reached coverage over time, on the other hand, often follows similar patterns as for SafireFuzz, just significantly later in time. Given that both frameworks use the same fuzzing backend, this is unsurprising.

SafireFuzz vs Fuzzware. SafireFuzz outperforms Fuzzware in terms of execution speed and uncovers more basic blocks except for the WYCINWYC and P2IM Drone samples. On 7 of the targets, Fuzzware performed signifi-

[3] Note that we deliberately refrain from reporting *paths*, as the metric is not well-defined and considered obsolete. In particular, the definition of unique input and the corresponding execution path differs widely across different fuzzers. Furthermore, the number of paths and the achieved code coverage do not necessarily correlate.

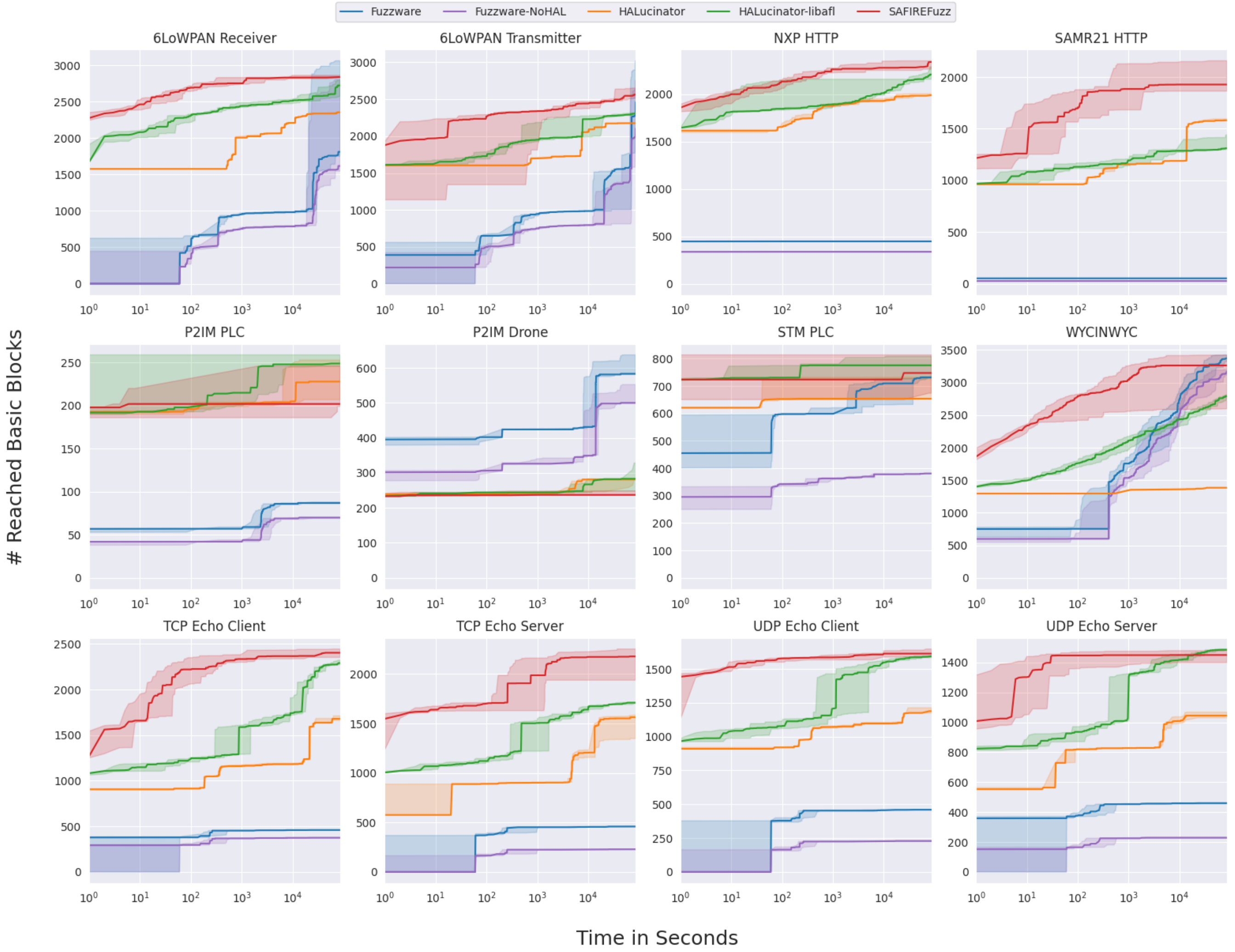


Figure 3: Coverage over time for SAFIREFUZZ, HALucinator, and Fuzzware in different configurations. Shown are the median and 95% confidence intervals over five 24-hour runs for each target.

cantly worse than the other frameworks. We suspect that this is due to the differences in automation: we used Fuzzware's *genconfig* functionality for harness creation. As such, the auto-generated harnesses may encounter roadblocks early on in emulation even before reaching the main logic of the target. These roadblocks are circumvented in HALucinator and SAFIREFUZZ due to HLE-based hooking.

With the exception of the 6LoWPAN Receiver/Transmitter, ST PLC, and WYCINWYC for coverage, and SAMR21 for execution speed, the presented results show statistical significance under the Mann-Whitney U test. For the 6LoWPAN samples, the experiments with Fuzzware consisted of single runs which found a high number of uncovered basic blocks, even exceeding the amount found with SAFIREFUZZ. In the case of WYCINWYC, Fuzzware achieved similar coverage to SAFIREFUZZ during the latter part of the 24-hour runs.

In terms of execution speed the Fuzzware barely reaches any blocks in the SAMR21 sample, and, thus, terminates execution early leading to very high throughput. However, SAFIREFUZZ provides similar throughput *while* discovering basic blocks and exercising large parts of the target.

SAFIREFUZZ vs Fuzzware (NoHal). When disregarding basic blocks located in HAL functionality, the reached coverage of Fuzzware decreases slightly. Unsurprisingly, other aspects of this experiment follow the same patterns as for Fuzzware without modifications.

However, one striking observation of this experiment is that even when hooking large parts of the HAL to provide firmware functionality, only a couple of hundred basic blocks are actually cut off from the target. Consequently, HLE-based approaches are losing less potential insight into the target than one might expect. At the same time, in the cases Fuzzware generates competitive coverage, a comparable amount of basic blocks is only found after a significant amount of time. While this can be partially attributed to the different seeds, we speculate that Fuzzware also spends a significant amount of time blocked in HAL functionality before reaching the main logic of a target.

6.3 Performance Analysis

Results of fuzzing the different firmware targets with our framework show a strong correlation between execution speed and found objectives (i.e., crashes and timeouts). For instance, when fuzzing the 6LoWPAN Receiver target, four runs achieved 559 executions per second with 18197 objectives on average. The outlier run only produced 3666 objectives and ran at 1536 executions per second. Similar results can be observed for all tested targets, although other factors, such as input length and validity correlating with path complexity, influence performance, too.

Throughput drastically decreases once the fuzzer finds its first crash. Every time our target process crashes or timeouts, the process has to be restarted and the engine has to perform all the heavy lifting again. Analysis and rewriting of basic blocks, especially (dis-)assembling instructions, are magnitudes more expensive than executing the firmware in the hot path, where most time is spent in the target and not in the engine. To estimate the performance penalty imposed by restarts, we created a microbenchmark using the WYCINWYC firmware. The first execution this firmware on a valid XML input requires 1.06 seconds averaged over ten runs. Subsequent runs on the same input execute at 6100 resets per second, i.e., they require only 0.00016 seconds. While there are multiple ways of addressing the problem of expensive, recurring restarts (e.g., snapshotting), we do not consider it to be a major problem since it is unusual for real-life fuzz campaigns to contain hundreds of crashes.

Fuzzing experiments with our framework show a relatively high variety in explored paths and executed basic blocks as well as in objectives and hence number of executions. This is presumably due to non-determinism in the used fuzzer backend. The divergence decreases in targets with few or no crashes. The partially drastic increase in covered basic blocks compared to other frameworks can be explained by the increased amount of tested inputs over 24 hours. That the increase in coverage is not linear is expected, too, as uncovering linearly more new parts of a program requires exponentially more executions [14]. This makes new ways of enhancing fuzzing performance, such as proposed with SafireFuzz, even more important.

Firmware	Minimized Crashes
WYCINWYC	16
SAMR21 HTTP	2
6LoWPAN Receiver	93
6LoWPAN Transmitter	27
P2IM PLC	14
STM PLC	325
JPEG Decoder	2
STM32Sine	1

Table 4: Crashes found in targets under test. We minimized crashes with AFL's `cmin`.

6.4 Vulnerabilities

During our experiments, we collected the objectives found with SafireFuzz. Additionally, we created harnesses for two new firmware samples. We report the minimized crashes in Table 4 and highlight noteworthy crashes in the following.

WYCINWYC. This firmware is intended as a benchmark for firmware fuzzing, assessing a fuzzer's capability to reach vulnerable code paths and, more importantly, to detect the fault by incorporating artificial vulnerabilities. It exposes five synthetically inserted memory corruptions within an XML parser, each corresponding to a different vulnerability type. SafireFuzz found all five bug classes, demonstrating our framework's ability to detect various kinds of corruptions. Our drop-in allocator replacement allowed us to find double free by keeping track of used and freed memory pointers. By making use of guard pages and relying on the host system's MMU, we can uncover segmentation faults and even heap overflows more reliably. For on-system fuzzing setups, identifying such memory corruptions is often more difficult, as many embedded devices do not have an MMU, and available emulators commonly rely on overhead-inducing SoftMMUs.

6LoWPAN Receiver/Transmitter. We re-discovered multiple vulnerabilities in Contiki-NG [23] originally found by HALucinator embedded in the 6LoWPAN Receiver target. Most notably, an out-of-bounds write in the data subsection that can be used for PC control, hence achieving Remote Code Execution (CVE-2019-8359). Other bugs include an integer overflow in the 6LoWPAN fragment processing, leading to a buffer overflow and, in turn, access to unmapped memory, crashing the firmware (CVE-2019-9183). Remarkably, in contrast to prior work, our approach *also* found a path triggering CVE-2019-8359 on the Transmitter. This demonstrates that SafireFuzz is capable of finding bugs not detected by prior work that fuzzed the same firmware (HALucinator [20], Fuzzware [46] and uEmu [51]).

JPEG Decoder. To test whether SafireFuzz can find bugs in additional firmware, we followed HALucinator's approach and compiled an example application. In particular, we target an application using LibJPEG [8] to decode and visualize an image embedded on an SD card inserted in the device.

Our fuzzing campaign found two previously unknown vulnerabilities. The first one is a segmentation fault caused by a critical error routine that, instead of terminating the program after beginning to parse a corrupted input image, falls through silently. Subsequently, no checks are in place to avoid accessing and dereferencing pointers in uninitialized structs in memory. We traced back the second vulnerability to missing bounds checks in the color conversion function. Output buffers have hard-coded sizes but the faulty routine uses the decoded image's width to iterate over scanlines and write to the buffer, heavily exceeding the stack-located buffer's limits.

STM32 Sine. The second additional target we test is open-source firmware for electric motor inverters [9]. During our fuzzing experiment, we explore a substantial part of the terminal interface which parses and processes various commands to change hardware-internal parameters via CAN bus communication. SAFIREFUZZ finds a crash related to updating certain parameter enumerations in the CAN configuration. An interface ID is retrieved from memory and used as an offset into memory. Corrupting this value leads to arbitrary memory writes. However, at the time of writing, we could not confirm whether this crash is a true positive, as part of its root cause lies in the hardware configuration, which may be reported wrongly by our HAL hooks.

7 Discussion

Performance of Initial Run. Our approach is very fast during fuzzing, once all blocks have been translated dynamically. However, most of the firmware's basic blocks are unknown during the first few executions, the fuzzer spends a majority of time inside the engine, reassembling and caching. Early executions hence run orders of magnitude slower than subsequent executions. Since each restart resets SAFIREFUZZ's caches, a further performance improvement would be to not exit on timeouts, or to write the cache to shared memory or disk. When receiving a `SIGALRM` signal, instead of restarting the whole process, the engine could just report a timeout exit code to LibAFL and manually reset the execution state. Resuming on a crash, e.g., a `SIGSEGV`, in a similar fashion is not easily possible. The run may have tampered with and corrupted our in-process state, as firmware execution happens in the same process space of the engine. Thus, any undefined behavior could also influence the engine's state or code. To tackle this issue, the caches could be tracked outside the current process, similar to the implementation of qemuafl [26]. Moving to a fully static rewriting is another option that, however, won't necessarily benefit fuzzing performance: the reassembly time is merely moved to the beginning, the resulting binary should not be faster overall, assuming we use the same techniques. On top, we lose valuable runtime information available during dynamic instrumentation.

Snapshotting. Since we target embedded firmware, we did not implement memory snapshots. The technique nowadays is increasingly applied in fuzzing. For it, a memory snapshot might be taken after a firmware's boot-up process, which would allow the engine to skip this part, and fast resets could be used instead of full restarts. For our use-case, boot-up routines of the firmwares investigated in the course of this work consisted of only a couple of hundred instructions. Since this is a very small amount of code, we decided that the additional overhead of snapshotting and memory resets would simply not be worth it. However, it could be interesting to evaluate in the future, as it allows for stateful fuzzing [39] and could allow faster resets upon crash.

Manual Effort. While our tool imposes the same restrictions as other HLE-based rehosting engines when it comes to adapting a new target to the system, the barrier to entry decreases over time as the potential reusability of user-provided hooks in this ecosystem is enormous. Many common and popular embedded platforms share their HALs and hooks for them need to be implemented only once: In the case of firmware targeting STM32 boards, we implemented a total of 18 generic HAL functions, whereas no function was used in less than two targets, and typical targets use up to 10 of these functions. Moreover, due to the plethora of existing HLE-based systems, readily available HAL stubs already exist. In terms of peripheral management and function-hooking capabilities, our engine offers functionalities compatible with many other HLE-based systems. For our experiments, we ported multiple HAL-emulating hooks from HALucinator's Python implementation to Rust without much effort. The authors of HALucinator [20] also argue that, while this method requires some manual effort, this allows HLE-based approaches to handle firmware automated systems such as P2IM [25] cannot.
In this work, we mainly focussed on improving execution speeds. In the future, reducing the required manual effort to adapt a new target to our fuzzer could be a worthwhile goal. Automatically identifying and hooking HAL functions would facilitate the analysis of a broader spectrum of firmware. Pairing our near-native rehosting approach for high performance with Scharnowski et al.'s MMIO-modeling technique [46] for increased generality seems extremely promising.

Hardware Platforms. Many emulation-specific techniques we employed for increased performance during fuzzing are adaptable to other domains or even CPUs architectures. Supporting additional ISAs, such as RISC-V, is mostly an engineering effort by extending the dynamic rewriting part of the engine with new translation passes. Yet, one of the core concepts of our approach is fundamental to the ARM environment: We exploit the fact that on the one hand, a large portion of the world's embedded software offering large and interesting attack surface runs on MCUs with ARMv7-M cores, and on the other hand, high-power commodity ARMv8 CPUs are widely available implementing the ARMv7-A instruction set as part of the AArch32 execution mode. Natively executing large portions of instructions of software compiled for very low-powered devices on vastly more powerful CPUs is to this extent unique to ARM. Running our framework on even more potent ARM cores is the logical next step. Development and testing were conducted on comparatively low-performance CPUs. Apple recently made powerful AArch64-based cores widely available and popular by introducing the M1 line [2]. However, in our experiments, we confirmed that 32-bit ARM support is not available in M1 chips and ARMv8 implementations generally seem to increasingly discontinue support. 32-bit support is, however, still supported on a wide range of products, from cheap development boards (e.g., the Cortex-A72 cores embedded on a Raspberry Pi 4[4]) over high-end consumer products (e.g., the Cortex-X1 cores used in 2022 Thinkpad X13s) and modern server-grade CPUs such as Ampere eMAG processors.

8 Related Work

Dynamic Binary Rewriting. Using dynamic binary rewriting to create a virtual execution environment for other software is a well-known concept. For instance, the original VMWare Workstation implementation [15] provided virtualization capabilities for x86 systems via system-level x86-to-x86 translation and a trap-and-emulate approach for sensitive operations. Similarly, QEMU [13] allows the emulation of different hardware platforms via dynamic binary translation. However, none of these approaches is tailored toward enabling low-level firmware fuzzing. Minor trade-offs in performance are accepted by design, and hardware accesses from the guest may require complex emulation back-ends. In contrast, SAFIREFUZZ's near-native rehosting approach enables running of code targeting an embedded ISA variant on a more powerful host with a different ISA variant, as long both variants belong to the same family (e.g., ARMv6-M and ARMv8-A).

Nonetheless, various frameworks explored binary rewriting for fuzzing, such as FЯIDA's Stalker mode [29] or AFL++'s Qemu- and Unicorn mode [26]. While these frameworks aim to provide optimized rewriting techniques to lower the runtime overhead, none of them considered the possibility of near-native rehosting. AFL++'s Qemu- and Unicorn lift the binary code to TCG, its intermediate representation, before applying instrumentation, and Frida requires that the ISA of the fuzzed target matches the one of the host.

Static Binary Rewriting. Recently, different static rewriting approaches for fuzzing have been proposed. Frameworks like *retrowrite* [21], *StochFuzz* [49], or *ZAFL* [43] move large parts of the one-time rewriting cost to a static offline phase. As a result, rewriting during run time is kept to a minimum or eliminated completely. While these approaches reach competitive performance compared to fuzzing via source-based instrumentation, none of these frameworks enable firmware fuzzing at the time of writing. All of them focus on either the x86 or AArch64 ISA, and have strong assumptions on the layout of the target binary, such as a clear distinction between code and data sections or position-independent code. We note that these assumptions which enable efficient static rewriting are rarely applicable to binary firmware, which is why we adopted a dynamic rewriting approach for SAFIREFUZZ.

Rehosting. In recent years, rehosting [24, 47] enabled fuzzing for various types of embedded systems, ranging from Linux-based IoT devices [36, 50] over wireless chipsets [33, 40, 45] to deeply-embedded devices with monolithic firmware [16, 20, 25, 31, 38, 46, 51]. SAFIREFUZZ draws direct inspiration from these frameworks and prototypes, especially from HAL-based rehosting approaches such as *HALucinator* [20] and *Para-Rehosting* [38]. However, in comparison to SAFIREFUZZ, most prior rehosting approaches focus on the creation of emulation environments for target firmware, rather than investigating possibilities for highly-efficient fuzzing solutions.

The most notable exceptions are *FirmAFL* and *Fuzzware*. *FirmAFL* aims to improve fuzzing efficacy for Linux-based firmware, by fuzzing single applications with QEMU's user mode emulator while selectively using full-system emulation to provide additional runtime context when needed. *Fuzzware*, on the other hand, targets monolithic firmware and integrates the fuzzer into the peripheral-modeling process while using local dynamic symbolic execution to narrow down the possible input space. While both solutions provide additions to firmware fuzzing, they both rely on a QEMU-based emulation engine and, unlike SAFIREFUZZ, do not explore an alternative low-overhead binary rewriting approach.

Concurrent to our work, MetaEmu [18] and ICICLE [19] aim to advance the state-of-the-art by broadening the range of rehostable architectures. As opposed to SAFIREFUZZ's near-native rewriting, both frameworks make use of Ghidra's processor and instruction set definitions to automatically derive virtualized execution environments. MetaEmu can also simultaneously rehost and analyze multiple connected targets. Although they do focus on performance and implement multiple IR optimization passes, they did not benchmark their approach against real-world targets from previous work. Unlike SAFIREFUZZ, they only show that they slightly outperform Unicorn on a few micro benchmarks. While ICICLE puts the focus on fuzzing, their main contribution is effective architecture-agnostic instrumentation. In contrast to SAFIREFUZZ, their framework - based on just-in-time compiled P-Code and a SoftMMU - does not fundamentally rethink emulation and achieves performance on par to Unicorn.

[4] During the development of SAFIREFUZZ, we confirmed that the framework runs on a Raspberry Pi 4 and achieves highly competitive performance.

9 Conclusion

In this work, we investigated the possibility of improving recent approaches for binary firmware fuzzing. Our engine, termed SAFIREFUZZ, leverages HAL-level hooking and dynamic binary rewriting for rehosting low-level ARM Cortex-M firmware onto more powerful ARM Cortex-A systems.

We evaluated SAFIREFUZZ by implementing fuzzing harnesses for the state-of-the-art firmware suite for HAL-based rehosting approaches. Our performance analysis shows that SAFIREFUZZ can provide a 690x throughput increase on average and a 30% improvement of basic block coverage compared to the state of the art over 24h fuzzing campaigns.

Overall, SAFIREFUZZ demonstrates that emulation efficiency is an important factor when designing rehosting systems with the ultimate goal to fuzz test embedded device firmware. We hope that the insights of our work will form the basis for faster firmware fuzzers in the future.

Availability

The source code of SAFIREFUZZ, as well as all evaluation harnesses and experiment code, is publicly available at: https://github.com/pr0me/SAFIREFUZZ.

Coordinated Disclosure

We disclosed the previously unknown vulnerabilities discussed in Section 6.4 to the maintainer of the STM32 Sine project and to the ST's Product Security Incident Response Team (ST PSIRT).

Acknowledgements

This work was supported by the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No. 101019206, the Dutch Ministry of Economic Affairs and Climate through the AVR program (Memo project) and the Dutch Science Organization NWO through projects Theseus and NWA ORC Intersect.

References

[1] American Fuzzy Lop Fuzzer. https://github.com/google/AFL. Last Accessed: 07.02.2022.

[2] Apple M1 Chip. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/. Last Accessed: 21.02.2022.

[3] ARM7TDMI Technical Reference Manual: The Thumb instruction set. https://developer.arm.com/documentation/ddi0210/c/CACBCAAE. Last Accessed: 21.02.2022.

[4] Capstone: Disassembler Framework. https://www.capstone-engine.org/. Last Accessed: 17.02.2022.

[5] Ghidra: Reverse Engineering Suite. https://ghidra-sre.org/. Last Accessed: 7.02.2023.

[6] hal-fuzz Github Repository. https://github.com/ucsb-seclab/hal-fuzz. Last Accessed: 06.02.2022.

[7] Keystone: Assembler Framework. https://www.keystone-engine.org/. Last Accessed: 17.02.2022.

[8] LibJPEG Decoder Firmware. https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/STM324x9I_EVAL/Applications/LibJPEG/LibJPEG_Decoding. Last Accessed: 07.02.2023.

[9] OpenInverter Firmware: stm32-sine. https://github.com/jsphuebner/stm32-sine. Last Accessed: 07.02.2023.

[10] UnicornAFL: A Bridge between AFL++ and the Unicorn Emulator. https://github.com/AFLplusplus/unicornafl. Last Accessed: 10.02.2022.

[11] Improving afl's qemu mode performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/, 2018. Last Accessed: 13.12.2022.

[12] Aspencore. Embedded systems market study, 2019.

[13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical Conference, 2005.

[14] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, 2020.

[15] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. ACM Transactions on Computer Systems (TOCS), 2012.

[16] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In Annual Computer Security Applications Conference (ACSAC), 2020.

[17] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-isa system mode emulation. In ACM Conference on Virtual Execution Environments (VEE), 2014.

[18] Zitai Chen, Sam L Thomas, and Flavio D Garcia. Metaemu: An architecture agnostic rehosting framework for automotive firmware. In ACM Conference on Computer and Communications Security (CCS), 2022.

[19] Michael Chesser, Surya Nepal, and Damith C. Ranasinghe. Icicle: A re-designed emulator for grey-box firmware fuzzing. In International Symposium on Software Testing and Analysis (ISSTA), 2023.

[20] 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 USENIX Security Symposium, 2020.

[21] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In IEEE Symposium on Security and Privacy (S&P), 2020.

[22] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020.

[23] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In IEEE International Conference on Local Computer Networks, 2004.

[24] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2021.

[25] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In USENIX Security Symposium, 2020.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In USENIX Workshop on Offensive Technologies (WOOT), 2020.

[27] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In ACM Conference on Computer and Communications Security (CCS), 2022.

[28] Andrea Fioralldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. Registered report: Dissecting american fuzzy lop - a fuzzbench evaluation. In 1st International Fuzzing Workshop (FUZZING), 2022.

[29] Frida. Stalker. https://frida.re/docs/stalker/, 2020. Last Accessed: 13.12.2022.

[30] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images. In IEEE Symposium on Security and Privacy (S&P), 2022.

[31] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In Symposium on Recent Advances in Intrusion Detection (RAID), 2019.

[32] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of Real-World TrustZone software using emulation. In USENIX Security Symposium, 2020.

[33] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware. In Symposium on Network and Distributed System Security (NDSS), 2022.

[34] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In USENIX Security Symposium, 2021.

[35] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral proxying supported embedded code testing. In ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2014.

[36] 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 Annual Computer Security Applications Conference (ACSAC), 2020.

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

[38] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From library portability to para-rehosting: Natively executing open-source microcontroller oss on commodity hardware. In Symposium on Network and Distributed System Security (NDSS), 2021.

[39] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. Fitm: Binary-only coverage-guided fuzzing for stateful network protocols. In Workshop on Binary Analysis Research (BAR), 2022.

[40] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: Baseband sanitized fuzzing through emulation. In ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec), 2020.

[41] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering, 47, 2021.

[42] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In Symposium on Network and Distributed System Security (NDSS), 2018.

[43] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In USENIX Security Symposium, 2021.

[44] Tobias Pfeffer, Paula Herber, Lucas Druschke, and Sabine Glesner. Efficient and safe control flow recovery using a restricted intermediate language. In IEEE Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2018.

[45] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In USENIX Security Symposium, 2020.

[46] 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 USENIX Security Symposium, 2022.

[47] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. ACM Computing Surveys (CSUR), 2021.

[48] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In Symposium on Network and Distributed System Security (NDSS), 2014.

[49] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In IEEE Symposium on Security and Privacy (S&P), 2021.

[50] 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 USENIX Security Symposium, 2019.

[51] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In USENIX Security Symposium, 2021.

[52] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In ACM Conference on Computer and Communications Security (CCS), 2022.

A Appendix

A.1 Comparison to other Papers

Firmware	SafireFuzz			HALucinator - Paper			Para-Rehosting		
	exec/s	*Time*	*Crashes*	*exec/s*	*Time*	*Crashes*	*exec/s*	*Time*	*Crashes*
WYCINWYC	3083.1	24h	16	17.92	24h	5	647.86	11h:43m	909
SAMR21 HTTP	2894.6	24h	2	22.92	19d:04h	273	902.95	12h:33m	219
NXP HTTP	5216.8	24h	0	154.5	14d:0h	0	1443.22	12h:39m	0
6LoWPAN RX	581.4	24h	93	18.84	1d:10h	3	–	–	–
6LoWPAN TX	1877.0	24h	27	15.3	1d:10h	0	–	–	–
P2IM Drone	7279.7	24h	0	11.8	9d:01h	0	–	–	–
P2IM PLC	772.1	24h	14	215	9d:01h	634	–	–	–
ST-PLC	7193.8	24h	325	3.73	1d:10h	27	2552.8	12h:15m	41
STM32 TCP Client	3401.3	24h	0	58.0	3d:08h	0	1092.4	12h:00m	58
STM32 TCP Server	2762.1	24h	0	56.7	3d:08h	0	1466.7	12h:00m	129
STM32 UDP Client	4636.7	24h	0	44.1	3d:08h	0	1245.0	12h:00m	65
STM32 UDP Server	3803.2	24h	0	66.7	3d:08h	0	902.3	12h:00m	16

Table 5: Throughput Comparison with experiments reported in HALucinator [20] and Para-Rehosting [38]. For SafireFuzz, we report values of the median run based on the number of executions. We minimized *Crashes* with AFL's `cmin` for our own experiments, for the other numbers it is not known whether or which minimization the authors applied.

In addition to our re-evaluation of different approaches, we also compare our approach to experiments described in other papers.

HALucinator. When compared to the numbers reported in HALucinator [20], gathered on a stronger CPU, we still achieve more than 200x on average (cf. Table 5). Consequently, we explore the targets much faster. The authors report that they found exactly five crashes in 612 paths. The divergence from our experiments, where HALucinator did not find a single crash, could be due to potentially less coverage, as we fuzzed the program on a weaker CPU and achieved fewer executions. During their 24-hour fuzzing campaign, the target was executed about 1.5 million times on a 12-core Xeon server. It is not stated whether the results were achieved by using a single or multiple cores per fuzz run.

Para-Rehosting. In this work, the proposed framework executed WYCINWYC about 27 million times within 12 hours. The authors report 3166 paths and 909 crashes, no absolute number of covered basic blocks is given. The median of crashes found with SafireFuzz on WYCINWYC is $25,653$ before minimization. While the achieved speedup compared to *Para-Rehosting* is existent but not extreme, the approach is no direct competitor as it requires compilable source code of the firmware. This often is a non-given in embedded security analysis. A comparison of execution speed can be found in Table 5.

P2IM. SafireFuzz offers a substantial speed-up over the P2IM technique on the tested firmware: Feng et al. report 32.7 and 17.2 executions per second for the PLC and Drone targets respectively, where we observe 772 and 7279 in the median case [25].

A.2 Special Control-Flow modifying Functions

Sometimes it is necessary to call unmodified functions in the firmware from within user-defined hooks, e.g., when we write an interrupt handler that resolves a callback. In emulators such as Unicorn, such behavior is handled by modifying the Program Counter in the emulation state. Returning from the user-defined hook will then resume execution at the specified address. Such modifications are not as trivial in our engine as there is no differentiation between firmware- and engine-PC. We handle this case by calling a dedicated *naked* function, Rust itself exposes no tailcall functionality. Naked functions do not incorporate any Rust-intrinsic function pro- or epilogues after compilation but consist of a single, pure assembly block. An example of such a function can be seen in Listing 1. The target function expects two parameters, the third one is used to supply the address of the target function. Upon finishing, it returns directly to the user-defined hook and Rust can perform its normal clear-up, dropping variables and correctly adjusting the stack frame.

Another case where special handling was necessary, are the GCC-specific thumb switch cases. They expose normal switch-case or table branch functionality but they work on making assumptions about the Link Register and directly modifying it. Naturally, this cannot work in our rewritten code site, where instructions are shifted by non-deducible amounts from their original addresses. As we expect this not to be the only time when a function might require knowledge of the LR, we introduced a saving mechanism. On every BLX, the address of the original callsite is stored at a globally-known memory location which then can be accessed by the callee. For the GCC switch-case functions, we simply hook and replace the original functions with naked functions, performing the same set of arithmetic modifications to calculate the new target address on the stored LR instead of using the register content.

```rust
#[naked]
unsafe extern "aapcs" fn _call_netif_input(
    _rx_pbuf_ptr: u32,
    _ethernet_netif_ptr: u32,
    _netif_input_cb_addr: u32,
) -> u32 {
    asm!("mov␣pc,␣r2", options(noreturn));
}
```

Listing 1: Example of naked function used for tailcalls from within user hooks.

A.3 Functions hooked at HAL-level

Function Name	System	Used in			
		WYCINWYC	*SAMR21 HTTP*	*6LoWPAN*	*P2IM PLC*
`malloc`	general	✓	✓	✓	✓
`realloc`	general	✓	✓	✗	✗
`free`	general	✓	✓	✓	✓
`puts`	general	✓	✗	✗	✓
`HAL_GetTick`	STM32	✓	✗	✗	✗
`HAL_RTC_GetDate`	STM32	✓	✗	✗	✗
`HAL_RTC_GetTime`	STM32	✓	✗	✗	✗
`serial_putc`	STM32	✓	✗	✗	✗
`serial_getc`	STM32	✓	✗	✗	✗
`mbed::Stream::write`	STM32	✓	✗	✗	✗
`mbed::Stream::read`	STM32	✓	✗	✗	✗
`rtc_write`	STM32	✓	✗	✗	✗
`rtc_read`	STM32	✓	✗	✗	✗
`HAL_SYSTICK_Config`	STM32	✗	✗	✗	✓
`HAL_UART_Receive_IT`	STM32	✗	✗	✗	✓
`HAL_UART_Transmit`	STM32	✗	✗	✗	✓
`HAL_UART_IRQHandler`	STM32	✗	✗	✗	✓
`millis`	Arduino	✗	✗	✗	✓
`HardwareSerial::read`	Arduino	✗	✗	✗	✓
`HardwareSerial::write`	Arduino	✗	✗	✗	✓
`HardwareSerial::available`	Arduino	✗	✗	✗	✓
`usart_write_wait`	SAM R21	✗	✓	✓	✗
`ethernetif_input`	SAM R21	✗	✓	✗	✗
`ksz8851snl_low_level_output`	SAM R21	✗	✓	✗	✗
`uip_tcpchksum`	RF233	✗	✗	✓	✗
`uip_udpchksum`	RF233	✗	✗	✓	✗
`rf233_on`	RF233	✗	✗	✓	✗
`rf233_off`	RF233	✗	✗	✓	✗
`i2c_master_read_packet_wait`	RF233	✗	✗	✓	✗
`trx_sram_read`	RF233	✗	✗	✓	✗
`trx_frame_read`	RF233	✗	✗	✓	✗
`trx_frame_write`	RF233	✗	✗	✓	✗
`trx_reg_read`	RF233	✗	✗	✓	✗
`trx_reg_write`	RF233	✗	✗	✓	✗
`clock_init`	Contiki-OS	✗	✗	✓	✗
`clock_time`	Contiki-OS	✗	✗	✓	✗
`clock_seconds`	Contiki-OS	✗	✗	✓	✗

Table 6: Functions hooked at HAL-level for four exemplary targets.