

EmbedFuzz: High Speed Fuzzing through Transplantation

FLORIAN HOFHAMMER, EPFL, Switzerland

QINYING WANG, EPFL, Switzerland and Zhejiang University, China

ATRI BHATTACHARYYA, EPFL, Switzerland

MAJID SALEHI, EPFL, Switzerland and Nokia Bell Labs, Belgium

BRUNO CRISPO, University of Trento, Italy

MANUEL EGELE, Boston University, USA

MATHIAS PAYER, EPFL, Switzerland

MARCEL BUSCH, EPFL, Switzerland

Dynamic analysis and especially fuzzing are challenging tasks for embedded firmware running on modern low-end Microcontroller Units (MCUs) due to performance overheads from instruction emulation, the difficulty of emulating the vast space of available peripherals, and low availability of open-source embedded firmware. Consequently, efficient security testing of MCU firmware has proved to be a resource- and engineering-heavy endeavor.

EmbedFuzz introduces an efficient end-to-end fuzzing framework for MCU firmware. Our novel *firmware transplantation* technique converts binary MCU firmware to a functionally equivalent and fuzzing-enhanced version of the firmware which executes on a compatible high-end device at native performance. Besides the performance gains, our system enables advanced introspection capabilities based on tooling for typical Linux user space processes, thus simplifying analysis of crashes and bug triaging. In our evaluation against state-of-the-art MCU fuzzers, EmbedFuzz exhibits up to eight-fold fuzzing throughput while consuming at most a fourth of the energy thanks to its native execution.

CCS Concepts: • **Security and privacy** → **Systems security; Embedded systems security**; • **Computer systems organization** → **Firmware; Embedded software**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Embedded Systems, IoT, Rehosting, Fuzzing, Arm Cortex-M

ACM Reference Format:

Florian Hofhammer, Qinying Wang, Atri Bhattacharyya, Majid Salehi, Bruno Crispo, Manuel Egele, Mathias Payer, and Marcel Busch. 2024. EmbedFuzz: High Speed Fuzzing through Transplantation. 1, 1 (December 2024), 27 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Authors' addresses: **Florian Hofhammer**, florian.hofhammer@epfl.ch, EPFL, Station 15, Lausanne, 1015, Switzerland; **Qinying Wang**, qinying.wang@epfl.ch, EPFL, Station 15, Lausanne, 1015, Switzerland and Zhejiang University, Hangzhou, China; **Atri Bhattacharyya**, atri.bhattacharyya@epfl.ch, EPFL, Station 15, Lausanne, 1015, Switzerland; **Majid Salehi**, majid.salehi@nokia-bell-labs.com, EPFL, Station 15, Lausanne, 1015, Switzerland and Nokia Bell Labs, Antwerp, Belgium; **Bruno Crispo**, bruno.crispo@unitn.it, University of Trento, Via Sommarive 14, Povo, 38123, Italy; **Manuel Egele**, megele@bu.edu, Boston University, Photonics Building, 8 St. Mary's Street, Room 337, Boston, Massachusetts, USA; **Mathias Payer**, mathias.payer@nebelwelt.net, EPFL, Station 15, Lausanne, 1015, Switzerland; **Marcel Busch**, marcel.busch@epfl.ch, EPFL, Station 15, Lausanne, 1015, Switzerland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2024/12-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

MCU-based embedded systems present a large attack surface due to their ubiquity, exposure to the internet, lack of Memory Management Unit (MMU)-based memory isolation, and development in memory-unsafe languages like C/C++. Vulnerabilities in MCUs used in devices like fitness trackers or smart light bulbs [25, 39, 71] as well as industrial applications [20] put end users, industry, and society at large at risk. Fuzz testing [28, 49] effectively uncovers memory safety bugs in a wide range of software, from user space software over Operating System (OS) kernels to hypervisors [11, 24, 47, 61, 70]. However, this promising testing technique faces several challenges when targeting MCU firmware.

Performance. Running MCU fuzzing campaigns on server-grade hardware suffers from a performance penalty due to incompatible Instruction Set Architectures (ISAs). For example, firmware targeting MCU based on the Arm, AVR, or Xtensa ISAs need to be translated dynamically to the native ISA of server-grade x86 processors. Previous works [17, 23, 52, 60, 73, 75, 76] leverage QEMU [10] as a full-system emulator, consequently paying a hefty *emulation tax*.

Peripheral Emulation. Embedded system firmware heavily relies on the Input/Output (I/O) capabilities of peripherals attached to the corresponding MCU. Emulating peripherals in the re-hosting process either entails great engineering effort [22] or requires sophisticated systems for automatic detection and emulation of Memory-mapped Input/Output (MMIO) accesses [23, 76], which may introduce inaccuracies. Forwarding peripheral accesses to physical hardware [52, 73] hinders scalability due to hardware dependence and incurs performance overheads stemming from communication with the physical device.

Binary-only Firmware. Embedded system firmware for commercial platforms is seldom available as Open-Source Software (OSS). Consequently, (re-)compiling firmware with fuzzing-enhancing instrumentation as provided by off-the-shelf fuzzers such as AFL++ [24] is only possible for a small subset of firmware. Low-overhead binary instrumentation is thus an important requirement for effective fuzzing. The dynamic binary instrumentation as provided by QEMU [10], for example in AFL++’s QEMU mode, comes with the emulation tax as described above. Static binary instrumentation, i.e., instrumenting the binary before its execution, provides low runtime overhead and is ideal for introducing the required instrumentation for fuzzing.

Our solution. We present EmbedFuzz, a high-performance, scalable, and accurate fuzzer for embedded system firmware. EmbedFuzz alleviates the emulation performance penalty by introducing and employing *transplantation*. Transplantation (1) employs static binary rewriting on binary firmware images to make them compatible with an ISA implemented in high-performance Central Processing Units (CPUs) that are similar to the targeted embedded system, and (2) executes the appropriately prepared bare-metal firmware as a user space process alongside a transplantation runtime on server-grade processors implementing such an ISA. Through transplantation, EmbedFuzz leverages the native performance of the underlying fuzzing hardware while at the same time scaling horizontally with the typically high core count of such systems. More specifically, our EmbedFuzz prototype transplants MCU firmware targeting Arm Cortex-M MCUs into Linux user space processes executing on systems built around Arm Cortex-A processors. Based on the observations made by HALucinator [17], EmbedFuzz identifies high-level interfaces to peripherals through binary analysis, and builds on top of the principle of High-level Modeling (HLM) introduced by HALucinator to emulate these interfaces. Finally, EmbedFuzz enables the use of established tools for both fuzzing as well as generic analysis and instrumentation purposes. By running transplanted

firmware as a native user space process, EmbedFuzz can leverage OS APIs such as `ptrace` used by existing tooling.

The key technique in our work is *transplantation*, the process of moving bare-metal firmware into a user space process. This process is challenging due to architectural and environmental differences between a bare-metal MCU firmware and a common Linux user space process. To address these challenges, EmbedFuzz leverages (1) modifications to the binary through static binary rewriting, (2) execution in our user space runtime, and (3) similarity of the host ISA and the ISA of the targeted embedded system for native execution. In combination, EmbedFuzz through transplantation achieves

- up to eight-fold increased fuzzing performance thanks to native execution of firmware binaries,
- horizontal scalability with core count thanks to MCU hardware independence,
- reusability of existing tooling for common user space processes.

EmbedFuzz is available as open-source software at <https://github.com/HexHive/SURGEON>.

2 BACKGROUND

To provide the necessary background for EmbedFuzz' design and implementation, we briefly highlight the main differences between MCU-based embedded systems and general-purpose CPUs. We also introduce Hardware Abstraction Layers (HALs) typically used in the development of embedded system firmware, and showcase fuzz testing as a viable software testing and bug finding technique.

2.1 MCUs and General-purpose CPUs

While MCUs share features such as arithmetic and logic capabilities with other CPUs, they target different use cases than general-purpose CPUs found in smartphones, tablets, or desktop computers. Namely, MCUs are typically custom-tailored for low cost and energy consumption. Consequently, they run at several orders of magnitude lower clock speeds of tens to a few hundred megahertz instead of multiple gigahertz. Additionally, MCUs lack features such as MMUs altogether, or make other features such as CPU caches optional. The latter is purely a performance optimization, but the former is a crucial requirement for many general-purpose OSs to run on a CPU. For example, Windows, or macOS require an MMU by design. While GNU/Linux can also run in absence of an MMU, the memory isolation through virtual address spaces for processes requires an MMU to be present. Any of those OSs requires comparatively much more performance than provided by an MCU to provide a smooth user experience. As a result, the main use case for MCUs is to run bare-metal software that is either designed to run without any OS abstractions at all or built around embedded OS flavors such as FreeRTOS [2] or Arm Mbed OS [8].

As an example, Arm's Cortex-M processor family represents a group of 32-bit Reduced Instruction Set Computing (RISC) MCUs. Chips from that family can be found in industrial applications (e.g., Programmable Logic Controllers (PLCs) [20, 31] and robots), consumer devices (e.g., smart scales [25] and activity trackers [71]), and the Internet of Things (IoT) (e.g., smart light bulbs [39] and thermostats). On the other end of the spectrum, the Arm Cortex-A family of 64-bit general-purpose CPUs powers smartphones [57] as well as high-end laptops, desktop computers, and server platforms [4, 5].

2.2 Hardware Abstraction Layer (HAL)

Hardware Abstraction Layer (HAL) libraries are a common building block of MCU firmware. Such libraries provide abstractions for developers to interact with the target system's hardware while

hiding low-level implementation details and functionality such as MMIO address ranges for data exchange with peripherals. The provided interface is typically common across MCUs belonging to the same family or even across families sold by the same vendor. A single HAL library can thus cover a large number of distinct MCUs. Consequently, firmware developed with HAL libraries are less tightly tied to a certain instance of the physical hardware. For example, STMicroelectronics provides the STM32Cube Software Development Kit (SDK) that includes HAL libraries covering all 17 MCU families comprising over 1,200 different configurations of their Arm Cortex-M based microcontrollers [65, 66].

2.3 Re-hosting

Re-hosting [22] is the process of building a virtual environment for a given firmware image that sufficiently models the firmware’s hardware dependencies. The goal of this process is to design this virtual environment in a way that analyses conducted on the re-hosted firmware in this environment are representative of executions on the real hardware. Previous re-hosting approaches mainly focus on the difficulty of emulating peripherals [17, 23, 60, 76] but leverage off-the-shelf emulators such as QEMU [10] or Unicorn [58] for ISA emulation, thereby incurring emulation overhead.

Modeling peripherals in a re-hosting environment typically follows one of two main angles. Either accesses to memory-mapped peripheral registers in the MMIO address range are handled directly or High-level Modeling (HLM) is employed. HLM intercepts control flow in the firmware before the access to the actual MMIO region happens. This interception occurs at a pre-defined higher level, e.g., at the level of the HAL API. HLM, as introduced by HALucinator [17] and employed in other systems [62], allows peripheral modeling with the semantics provided by the HAL. Instead of requiring to accurately mimick complex real hardware’s behavior, only the HAL API’s behavior with known semantics needs to be replicated. A firmware’s application logic calling into HAL libraries is fully oblivious to the logic executing below the HAL. Consequently, both HLM and MMIO-based peripheral modeling in this case do not affect the semantics of the firmware’s application logic.

2.4 Grey-box Fuzzing

Fuzz testing or *fuzzing* is an automated testing technique to find bugs in software by repeatedly executing the Program Under Test (PUT) with new inputs [28, 49, 67]. Ideally, those inputs trigger different behavior in the PUT so that as much of the PUT’s Control Flow Graph (CFG) as possible is covered and bug locations are therefore visited. If a bug is triggered at such a location, the fuzzer detects the bug either through a crash of the PUT or through timeouts if execution stalls. In coverage-guided grey-box fuzzing, the PUT is instrumented to report back to the fuzzer when it reaches new locations. This feedback allows the fuzzer to identify increased code coverage, which in turn can be used for detecting and mutating interesting inputs to exercise the newly reached part of the code. As a result, this methodology has allowed fuzzers such as AFL++ [24] or Syzkaller [70] to unearth numerous bugs and vulnerabilities in both user space applications and OS kernels [34, 69].

3 FUZZING MCU FIRMWARE CHALLENGES

In this section, we detail the challenges of rehosting MCU firmware that we briefly mentioned in the introduction. Table 1 summarizes the comparison of different approaches.

Challenge 1: Representation of the Physical Address Space. MCU firmware statically links all code and global data. MCUs then map this firmware, its heap and stack as well as MMIO-mapped peripherals and control registers into a single flat address space. Those different memory regions

Table 1. Dynamically analyzing rehosted MCU firmware is challenging. This table shows how existing approaches address these challenges. E - Emulation, N - Native, O - badly supported, ● - well supported

Challenges		Hardware-in-the-loop	Pure Emulation	High-level Modeling	Transplantation
	E/N	E	E	N	
Phys. Addr. Space Repr.	E/N	E	E	N	
Architectural differences	E	E	E	E/N	
Peripheral Handling	N	E	E	E	
Insn Execution	E	E	E	N	
Speed+Scalability	O	●	●	●	
Introspection	●	●	●	●	

for ROM, RAM or MMIO are placed at fixed locations in this physical address space. Re-hosting solutions must account for the corresponding address space structure to accurately model the firmware's expected execution environment.

Hardware-in-the-loop, pure emulation, and HLM approaches all introduce a complex emulation layer. This layer imposes a software-based address translation for every access to the address space to compensate for any side effects. For example, accessing the MMIO region requires peripheral-specific handling of loads and stores, whereas accessing RAM requires only Memory Protection Unit (MPU) access permission checks. In summary, all accesses must respect the original MCU's physical address space layout.

Challenge 2: MCU Architectural Features. MCUs differ architecturally compared to a CPU's user space mode that re-hosting solutions typically execute in. Such differences include, e.g., execution modes, interrupt controllers (Arm's Nested Vectored Interrupt Controller (NVIC)), banked registers, or hardware-supported exception entry and return routines not present in a Linux process. Especially the latter presents a significant difference between Arm Cortex-M MCUs and an unprivileged Cortex-A user space context. Loading a value into the program counter register always triggers a branch to that location in user space, while loading a magic value (0xFFFFFFFF1 to 0xFFFFFFFFD) instead triggers a hardware-implemented exception return routine on Arm Cortex-M. Re-hosting must account for such features.

Previous hardware-in-the-loop [52, 73], HLM [17] or purely emulation-based [23, 60] re-hosting solutions commonly emulate the corresponding logic, managing complex CPU state solely in software. Other approaches trying to execute MCU firmware in user space ignore this challenge altogether [46, 62].

Challenge 3: Peripheral Handling. MCU firmware interacts with the outside world using peripherals. A re-hosting environment must handle peripheral accesses with varying degrees of accuracy, depending on the analyses to be conducted and how the results of those analyses map to the real hardware.

Hardware-in-the-loop approaches are commonly the most accurate due to their usage of "real" peripherals. Pure emulation provides high accuracy using the peripheral's MMIO interface. However, this approach suffers from significant engineering efforts to correctly implement the vast amount of available peripherals [22]. HLM leverages the HAL layer of peripherals to reduce the engineering

effort for peripheral emulation. However, as a consequence, this approach does not execute low-level peripheral specific code beneath the HAL layer. The HAL interface provides information about the semantics of data being passed in and out of the firmware from and to peripherals, allowing highly accurate modeling of those interfaces. Peripheral modeling based on symbolic modeling or automated peripheral behavior inference exposes a higher degree of automation but fails to guarantee the same level of accuracy.

Challenge 4: Correct Execution of Instructions. The execution of instructions must be functionally equivalent to the execution on the original MCU. While this generally does not pose an abundant problem, emulators and hypervisors may introduce inaccuracy when modeling the behavior of a physical CPU [3, 26, 50], impacting functional correctness of the re-hosting environment.

Challenge 5: Speed and Scalability. Dynamic analysis techniques require speed and horizontal scalability. For example, fuzzing benefits from fast and massively parallelized execution due to more trials being executed in less time.

Hardware-in-the-loop approaches emulate the target’s ISA and forward peripheral accesses to real hardware via debug probes. They require dedicated MCU hardware for each running firmware instance. Thus, these approaches lack scalability and suffer from a performance degradation due to the forwarding of accesses to the physical device. In contrast, pure emulation and High-level Modeling (HLM) approaches do not require dedicated hardware and scale horizontally. Unfortunately, these approaches pay a hefty emulation tax for running a foreign ISA on the host’s CPU.

Challenge 6: Introspection. Dynamic analysis requires flexible introspection capabilities. Hardware-in-the-loop, pure emulation, and HLM usually require presence of a hardware debugging interface or modification of the emulation layer, i.e., adding a debugger stub to enable attaching a debugger. Therefore, these approaches result in additional engineering effort, a restricted feature set in comparison to common interfaces for user space software, communication overhead, or a combination of the previous.

Challenges Summary. We designed EmbedFuzz to address these challenges. EmbedFuzz provides accurate modeling of MCU behavior paired with high execution speeds and simple horizontal scalability by combining accurate native hardware behavior with the ease of handling minimal Linux user space processes. As a result, our firmware transplantation approach, EmbedFuzz, outperforms related approaches when fuzzing MCU firmware.

4 EMBEDFUZZ DESIGN

Binary translation captures the idea of translating from one execution environment and ISA to a fundamentally different ISA and environment combination. For instance, it enables the execution of an Arm Cortex-M firmware on an x86-based machine leveraging an emulator. On the other hand, *direct execution* of a firmware on the intended MCU provides an analyst with firmware behavior exactly exhibiting the behavior of a real-world deployment.

We design and build EmbedFuzz based upon the insight that the Arm Cortex-M and Cortex-A ISAs overlap to a large extent. Based on this observation, we introduce *transplantation* as a middle ground between binary translation and direct execution. In transplantation, we leverage the semantic compatibility between the ISAs to execute the majority of binary code in a Cortex-M firmware without modification in a Linux user space process on a Cortex-A based system. We address remaining syntactic and semantic incompatibilities through a combination of static binary rewriting and a custom runtime, resulting in minimal changes to the firmware binary.

Transplantation entails two major advantages. First, EmbedFuzz achieves high execution speeds thanks to native execution of firmware code. Both binary translation and direct execution of

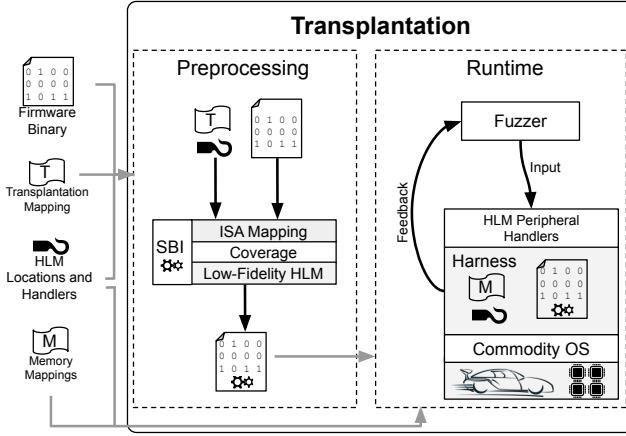


Fig. 1. The overview of EmbedFuzz’s approach to MCU firmware fuzzing. Our contributions are shown in gray.

firmware suffer from low execution speeds. The former requires an expensive architectural transplantation layer, while the latter experiences the low clock speeds of MCUs. Second, the significant overlap between MCU and CPU ISAs, and careful handling of the remaining differences provides EmbedFuzz with a means to replicate firmware’s behavior with high fidelity.

EmbedFuzz leverages those insights for *transplanting* embedded firmware, creating fuzzable user space binaries in its *transplantation phase*. EmbedFuzz handles peripheral accesses using High-level Modeling (HLM) in its *runtime phase*. Together, transplantation and HLM enable fast and accurate fuzzing while tackling the challenges discussed in Section 3. We refer to Figure 1 for an overview over EmbedFuzz’s components.

Fuzzing embedded firmware with EmbedFuzz requires a *transplantation phase* that transforms the firmware into a fuzzable user space binary through a sequence of four steps. During this transplantation phase, EmbedFuzz modifies the binary to *i*) address semantic differences between the ISAs of Cortex-M and Cortex-A CPUs (ISA mapping), *ii*) insert coverage instrumentation (coverage), *iii*) insert peripheral emulation code to directly replace simple HAL functions (low-fidelity HLM), and *iv*) insert branches to the HAL handlers in our runtime for complex HAL functions (high-fidelity HLM).

After the transplantation phase, EmbedFuzz’s runtime ensures correct execution of the prepared binary. Our custom loader maps the binary with a virtual memory layout matching the physical memory layout of the targeted MCU, thereby maintaining the validity of hardcoded addresses within the binary. During analysis with a general-purpose coverage-guided fuzzer, EmbedFuzz runs the firmware natively within a runtime which *i*) emulates architectural differences that cannot be addressed by static binary rewriting in the transplantation phase, and *ii*) emulates the complex behavior of peripherals with HLM. The runtime serves as the fuzzing harness, spawning a new process representing the firmware’s initial state and forwarding inputs from the fuzzer for each iteration. The runtime also monitors the process for anomalies (i.e., crashes and hangs) in order to return feedback information (code coverage, termination status) to the fuzzer.

In the following, we list the prerequisites for EmbedFuzz’s approach and detail the design of the *transplantation* and *runtime* phases.

4.1 System Model and Prerequisites

EmbedFuzz targets firmware that lacks a classic OS which would provide process isolation and virtual address space abstractions. Thus, EmbedFuzz assumes that the processes run on bare metal. While the firmware binary must be available, the source code of the firmware is not required. Compared to emulator-based re-hosting techniques, transplantation has the additional requirement that the high-end processor used for the fuzzing system must natively support the simpler MCU's ISA. This requirement is satisfied by a wide variety of commodity platforms, with an abundance of both embedded and high-performance processors based on Arm, PowerPC, or RISC-V. Our prototype targets MCU firmware developed for Arm Cortex-M. This architecture is the most widely-deployed processor family for 32-bit MCUs, and it is still actively used and deployed in new MCU designs [1, 21, 42, 59]. As detailed in Section 4.2, EmbedFuzz's design however is not limited to this architecture. Similar to HALucinator [17] and Safrefuzz [62], we assume that HAL libraries are available to the analyst. Cortex-M vendors provide open-source HAL libraries for their chips, with compilers and configurations. Due to the benefits provided by these HAL libraries, it is expected that most developers utilize them [17].

EmbedFuzz discovers software bugs and vulnerabilities in the firmware's code that can be triggered by unexpected inputs provided to peripheral interfaces, e.g., SPI, UART, or Ethernet. We leverage the full control over the inputs of such exposed interfaces to inject fuzzing input into the firmware.

4.2 Firmware Transplantation

The principle of *transplantation* is our key contribution to enable high-performance execution of embedded system firmware binaries. Transplantation requires ensuring correct execution of instructions as well as mimicking the address space layout and architectural behavior that the firmware expects. Transplantation relies on the insight that the Thumb-2 instruction set implemented by Cortex-M processors, which represent a dominant fraction of the embedded device market, is similarly implemented in Cortex-A processors popular in mobile and desktop CPUs. Therefore, transplantation enables EmbedFuzz to execute embedded firmware natively (i.e., without the cost of emulation) on Cortex-A processors significantly faster than their original embedded system. With desktop and server grade platforms incorporating up to 128 cores (e.g., for Ampere's Altra Max [4]), transplantation enables horizontal scaling of fuzzing campaigns. Both those factors help EmbedFuzz address Challenge 5 by obviating performance overheads of running MCU firmware either on-device or in an emulator. Native execution of firmware as a side effect also ensures correct execution of instructions on real hardware (Challenge 4).

Address space layout. Transplantation must preserve the address space layout of the firmware since embedded code is typically statically linked with hardcoded compile-time addresses. Consequently, the firmware expects code, data, and other memory regions such as MMIO regions at certain locations in its physical address space (Challenge 1). Our static binary rewriter accounts for this requirement by copying unmodified instructions and data into the same location in the resulting binary.

MCU environment modeling. As described in Challenge 2, a re-hosting solution must sufficiently replicate the real device's behavior for accurate dynamic analysis. While the bulk of the MCU's instructions can be natively executed on the fuzzing host, transplantation must account for the few instructions exhibiting different semantics on the two devices, including device-specific effects such as programming interrupts, execution mode changes or co-processor accesses. EmbedFuzz accounts for these instructions by substituting them inline with semantically equivalent instructions. In the

case where the inserted instructions would take up more space than the original instruction in the binary did, EmbedFuzz follows a trap-and-emulate approach and inserts traps to the runtime (see Sections 4.3 and 5).

In addition to handling differing instruction semantics, transplantation models MCU hardware features not available on the transplantation host system through additional architectural resources on high-performance CPUs to address Challenge 2. For instance, transplantation re-uses general purpose registers not employed in the firmware for modeling MCU-specific registers. EmbedFuzz leverages this observation to implement features such as the stack pointer register `sp` on Cortex-M being an alias to two banked stack pointers used in the interrupt and standard mode, respectively. We model this specific Cortex-M feature by mapping the stack pointer banks onto vector registers only present on Cortex-A and rewriting instructions accessing the corresponding registers accordingly.

Peripheral emulation. Transplantation must account for interactions with peripherals in order to correctly model I/O and interrupt behavior (Challenge 3). To this end, EmbedFuzz leverages HALucinator’s high-level modeling approach [17]. In the transplantation phase, EmbedFuzz inserts a branch to the corresponding emulation code for each identified HAL function. As an optimization, simple HAL functions such as peripheral initialization functions are replaced inline with appropriate emulation code by our binary rewriter (low-fidelity HAL).

Fuzzing enhancements. Finally, EmbedFuzz optionally inserts fuzzing-specific instrumentation, particularly edge coverage tracking, during static binary rewriting. EmbedFuzz inserts trampolines to the instrumentation into basic blocks with sufficient space for the corresponding branch instruction. EmbedFuzz does not use the trap-and-emulate approach to track the remaining basic blocks due to the high overhead of trapping. With this design choice of opportunistic instrumentation, EmbedFuzz trades performance for coverage accuracy in grey-box fuzzing campaigns. As we empirically demonstrate in Section 6, this design choice is a reasonable trade-off because the number of uninstrumented basic blocks in our dataset is low and the fuzzer still manages to generate coverage outperforming the competition.

4.3 Runtime Environment

EmbedFuzz’s runtime environment orchestrates the dynamic behavior required to correctly fuzz the firmware. First, the runtime sufficiently mimics the MCU’s behavior and handles emulation of complex instructions which trap in the prepared binary to ensure correct execution of the firmware. Second, the runtime contains the HAL handlers directly called by the firmware. Third, the runtime acts as a fuzzing harness for integration with general-purpose fuzzers.

Address space layout. EmbedFuzz’s runtime ensures a correct address space representation (Challenge 1) by loading the firmware into a Linux process’s virtual address space according to the layout dictated by the MCU’s physical address space and the original firmware binary.

MCU environment modeling. After loading the transplanted firmware binary, the runtime hands over control to the natively executing firmware. In order to correctly execute the firmware’s code and model the MCU’s features (Challenges 2 and 4), EmbedFuzz employs multiple techniques. First, the runtime handles traps issued during execution of the firmware in case instructions with different semantics between Arm Cortex-M and Cortex-A could not be substituted inline during the transplantation phase. Second, the runtime faithfully emulates MCU-specific behavior such as saving and restoring execution context onto and from the stack or switching of banked registers upon exception entry and return. For example, loading magic values into the program counter on Cortex-M while in an Interrupt Service Routine (ISR) triggers a hardware-supported exception return routine restoring context from the stack instead of an actual branch to the register value. In a

Cortex-A user space process, the CPU will interpret those magic values as addresses, try to execute code from that location, and raise a segmentation fault. By mapping branches to the corresponding software implementation of the exception return routine at the addresses corresponding to the magic values, EmbedFuzz avoids such false positive crashes and emulates the behavior expected by the firmware.

Since embedded firmware generally process external interrupt-driven inputs, the runtime must also correctly and reproducibly emulate interrupt behavior. The reproducibility is essential for the developer to successfully triage bugs. Consider, for example, a timer which interrupts the firmware at regular intervals in order to poll an attached Universal Asynchronous Receiver-Transmitter (UART) peripheral. The runtime must periodically interrupt the firmware and call the timer ISR to emulate this behavior. EmbedFuzz employs an interrupt scheduling approach similar to P2IM [23] or HALucinator [17], delivering interrupts in a round-robin fashion at a fixed interval. In contrast to the aforementioned systems, EmbedFuzz schedules interrupts through a virtual clock based on instruction counting instead of basic block counting.

Peripheral emulation. EmbedFuzz’s approach to peripheral emulation targets the higher-level HAL layer rather than the low-level I/O instruction layer that typically consists of MMIO accesses, Direct Memory Access (DMA) and interrupts. By leveraging the HLM principle introduced by HALucinator, we consequently allow replacing hardware peripherals with fully flexible software implementations ([Challenge 3](#)). In EmbedFuzz, the transplanted binary calls into the runtime instead of the firmware’s HAL. The runtime integrates these calls with the corresponding emulation code. Writing the handlers for each HAL function requires manual effort. However, this effort is acceptable as the cost amortizes across all firmware sharing the same HAL library. Additionally, EmbedFuzz permits reusing existing handlers from HALucinator [17]. These already existing handlers provide a baseline to an analyst and can still be adapted according to the use case.

Introspection. Since transplanted firmware runs inside a user space process, transplantation facilitates the use of the plethora of dynamic analysis tools developed for user space processes without restrictions or additional engineering requirements ([Challenge 6](#)). For example, unmodified fuzzers such as AFL(++) [24, 74] or memory leak checkers such as Valgrind [53] can be directly applied to a transplanted firmware. Often, similar tools for MCUs and emulation-based systems do not exist, or require additional steps or setup overhead. Resetting an MCU’s state between fuzzing iterations is slow and requires hardware debuggers with JTAG support, whereas the corresponding transplanted user space process can be replicated with a fork system call. Horizontal scaling for transplanted firmware amounts to spawning additional processes on multi-core systems ([Challenge 5](#)). In contrast, similar scaling with MCUs requires procurement of additional hardware and its orchestration.

5 IMPLEMENTATION

Our EmbedFuzz prototype implements firmware transplantation as described in the previous section for fuzzing firmware based on Arm’s Thumb-2 ISA. Here, we describe key details of the implementation of our EmbedFuzz prototype which we evaluate in [Section 6](#).

Code Additions. EmbedFuzz maps added runtime code (for instruction emulation, peripheral handling or fuzzing instrumentation) into an address range reserved for vendor code as per the Arm Manual [6]. The Vendor_SYS region, covering the address range 0xE0100000–0xFFFFFFFF, is unused in each of our analyzed firmware. Even if a vendor makes use of parts of this memory range, it provides ample space for storing any required code.

Transplantation. For EmbedFuzz, we designed a custom static binary analysis and instrumentation framework to transplant firmware and generate the fuzzing binary. EmbedFuzz transplants MCU firmware, replicating non-code sections and generating equivalent code for code sections using the methodology described previously: copying the majority of instructions, and either rewriting or emulating special instructions. Through this approach, EmbedFuzz ensures semantically correct firmware execution on an application processor, which is crucial for the accuracy of a transplantation-based re-hosting system.

We determined the subset of special instructions by manually comparing the Cortex-M and Cortex-A Thumb-2 instruction sets. We first identified syntactically different instructions by forcefully disassembling any possible two- and four-byte binary values that could constitute an instruction and then comparing their mnemonics. The results confirmed our assumption that the set of Thumb-2 instructions supported on Cortex-A CPUs is a superset of the one encountered in our Cortex-M firmware dataset. We then classified the instructions according to their behavior. While it is safe to assume that the class of basic integer arithmetic instructions such as add or sub behaves the same way on a Cortex-M and a Cortex-A chip, other classes such as interrupt-generating instructions (e.g., svc) require manual inspection. This one-time effort does not need to be repeated for further use of our system.

EmbedFuzz identifies three main categories of special instructions. The first category consists of instructions that generate software interrupts. For example, the execution of an instruction to change the privilege level (e.g., svc (supervisor call)) might behave differently on a high-end device. The second category contains instructions that change the state of the CPU and usually require the current execution mode to be privileged. An example for this category is enabling or disabling interrupt reception for the current core (e.g., cpsr{e,d} (change processor state)). In the third category, we find instruction-based I/O which grants access to CPU-external resources using specific instructions (e.g., mcr (move to coprocessor from Arm register)). Accesses to co-processors using explicit instructions instead of MMIO fall into this category. We provide further details on instructions requiring special handling in [Appendix A](#).

EmbedFuzz replaces special instructions with semantically equivalent instructions or, if not possible due to space constraints at the instrumentation location, with traps. The emulation code and trap handlers reside in the Vendor_SYS region. For trapping, we use the bkpt instruction, which causes the OS to raise a SIGTRAP signal to the runtime. The runtime in turn implements the corresponding signal handler. We uniquely identify each instruction (opcode and operands) to be emulated with the bkpt instruction's 8-bit immediate field, allowing the handler to dispatch to the correct emulation routine by reading this immediate field. For example, supervisor call instructions (svc) are translated to bkpt #1 instructions. Apart from specific instructions that cannot be replaced inline, we also replace infinite empty loops (i.e., a branch to itself) with a bkpt to fast-forward EmbedFuzz's virtual clock without stalling execution in such loops. We observe that firmware uses this pattern of an infinite loop for purely interrupt-based logic, i.e., the main control flow of the firmware depends on interrupts triggered by timers and other peripherals. By fast-forwarding our internal virtual clock, we immediately trigger the next interrupt while upholding the firmware logic's perception of time passed. This is in contrast to previous and concurrent work [60, 62], which do not detect and handle such loops waiting for interrupts.

While the bkpt approach limits EmbedFuzz to trap up to 256 different instruction types due to the size limitation of the immediate operand, we found this sufficient for our fuzzed firmware. An alternative implementation, adding complexity, can leverage a lookup table to determine the instruction replaced with a bkpt and differentiate based on the actual instruction instead of our bkpt operand. Thus, this limitation is a conscious simplifying implementation choice.

Interrupts and Exceptions. EmbedFuzz emulates asynchronous software, MCU and peripheral-driven interrupts by periodically calling the corresponding handler in firmware based on the programmed configuration in the interrupt vector table. Handlers for HAL functions configuring interrupts set up timers emulated in our runtime which deterministically and reproducibly fire interrupts based on the number of instructions executed. Of particular importance, EmbedFuzz emulates the configurable timer interrupt (SysTick) oftentimes leveraged for preemptive task scheduling in Real-Time Operating Systems (RTOSs). This approach allows EmbedFuzz to have reproducible fuzzing campaigns.

Importantly, EmbedFuzz also implements the Arm Cortex-M exception entry and return routines executed by hardware on a real MCU. EmbedFuzz stores and restores context information to/from the stack as the hardware would [6] and also emulates switching between the banked stack registers SP_main and SP_process. The latter can be achieved very efficiently by re-purposing the Cortex-A's double-precision floating point registers from d16 onward, which are not available on Cortex-M MCUs [6, 7]. Consequently, we replace the mrs/msr (move from/to special register) instructions accessing the banked stack pointers with a single vmov instruction in the binary rewriter, and switch stack pointers before/after an ISR implemented via two vmos for storing/restoring the corresponding banked version of the stack pointer. This feature is crucial for supporting RTOS-based MCU firmware, since task switching in RTOSs is commonly implemented by setting the stack and banked stack registers up in an interrupt routine in such a way that the hardware-implemented exception return routine switches to the desired stack and restores the corresponding context from the stack.

Fuzzing Enhancements. EmbedFuzz implements support for AFL++ by inserting trampolines to record coverage with basic-block granularity. Basic blocks are identified with Ghidra [27], and the corresponding coverage instrumentation is mapped into the Vendor_SYS region. For certain basic blocks, we need to replace multiple Thumb-2 instructions (typically 2 bytes) to fit the required 4-byte branch instruction. A minority of basic-blocks (~ 10%), such as one holding a single ret instruction, are too small to be instrumented, and are ignored. Note that, while reducing accuracy, ignoring some basic blocks does not overestimate the number of basic blocks covered. Consequently, coverage reported by our instrumentation is a lower bound on the actual coverage achieved.

Peripheral Handling. EmbedFuzz builds on HALucinator's [17] High-level Modeling (HLM)-based approach to emulating peripherals. We use any symbols exported by the firmware or fall back to HALucinator's LibMatch to identify HAL function calls and replace them with calls into the runtime's handler functions. Our runtime implements HAL handlers as C functions following the publicly documented function prototype for the corresponding HAL function. Thanks to the adherence to the function prototype and the standardized Application Binary Interface (ABI) in the Arm Architecture Procedure Call Standard (AAPCS) [9], our handlers can consequently access HAL functions' arguments and return values directly without the need of context modifications in between the invocation of the HAL function in the firmware and the execution of the corresponding handler.

Alternatively, our runtime links against libpython, allowing rapid prototyping of HAL handlers in Python and, more importantly, permitting reusing HALucinator's peripheral handlers directly as a baseline for further implementation. The embedded Python interpreter can be used to execute peripheral handlers written in Python in-process, thus not requiring a costly context switch to another instance of a Python interpreter. In contrast, related approaches rely on Linux's ptrace functionality or dispatches from QEMU into a different handler process [17] and therefore add expensive context switches for each HAL function call.

Smaller or simpler handlers, such as HAL_TIM_OsConfig used in the STM32 HAL library for MCU clock configuration, can be replaced inline in the firmware code region or simplified to an empty function. In this specific example, hardware clock configuration is irrelevant for EmbedFuzz since timers are emulated in the runtime instead of relying on hardware clocks, providing an analyst with fine-grained control over the firmware’s perception of time.

6 EVALUATION

We evaluate EmbedFuzz on real-world firmware and compare it against other MCU-based coverage-guided greybox fuzzers. We also conduct experiments to show that our transplantation approach is accurate, i.e., that EmbedFuzz does not cause crashes or other unexpected behavior which are not originating from actual bugs in the firmware. Our evaluation is guided by the following research questions that directly support our earlier claims.

RQ 1: Throughput. What is EmbedFuzz’s throughput (i.e., executions/sec) in comparison with other state-of-the-art coverage-guided greybox fuzzers for MCU firmware?

RQ 2: Coverage over time. What code coverage does EmbedFuzz achieve during fuzzing campaigns in comparison to the other systems we evaluate against?

RQ 3: Bug finding capabilities. Which bugs in our evaluation dataset are uncovered by EmbedFuzz and the other evaluated systems?

RQ 4: Computational resources. What is the impact of EmbedFuzz’s transplantation approach on computational resources required for fuzzing campaigns?

RQ 5: Transplantation fidelity. How does transplantation fidelity impact accurate high-performance dynamic analysis of MCU firmware?

RQ 6: Scalability. What is the amount of manual work required by an analyst to transplant a firmware with EmbedFuzz and how does this affect scalability to other firmware?

6.1 Experimental Setup

In our experiments, we use 10 real-world firmware samples presented in P2IM [23]. These are full-fledged MCU firmware and power various embedded devices such as PLCs, drones, and robots. They contain all the common MCU firmware components (e.g., interrupt handlers and system libraries) and collectively cover Arm Cortex-M3 and Cortex-M4 microcontrollers. To evaluate EmbedFuzz’s effectiveness and efficiency, we compare it with two state-of-the-art coverage-guided greybox fuzzers for MCU firmware: P2IM [23] and Fuzzware [60]. Unfortunately, we encountered multiple issues in the μ Emu [76] codebase¹ that prevented us from successfully running all the desired experiments. Based on several weeks of engineering efforts, some crashes in the S2E symbolic execution framework leveraged by μ Emu could be resolved thanks to correspondence with the author, other problems such as excessive memory usage causing out-of-memory situations still persist. Those problems are also documented in the project’s public GitHub repository and have been encountered independently by multiple users. In consequence, we do not consider μ Emu in our evaluation. We do not evaluate against HALucinator’s fuzzing component, hal-fuzz, and Safirefuzz [17, 62]. These two systems lack features modeling MCU behavior related to interrupts that transplantation provides and that are required for accurate re-hosting of our evaluation dataset. Consequently, the two systems are not compatible with our evaluation dataset. We detail this observation in a case study in Section 6.7.

¹git commit a191402f5704fc231a7590fad7bc118f4208616b

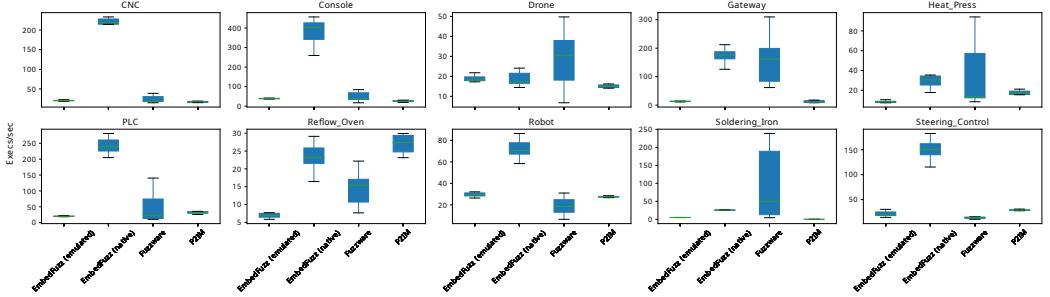


Fig. 2. Box plot of fuzzing executions per second (i.e., throughput) on real-world firmware binaries across ten 24 hour runs.

All experiments for transplanted firmware are conducted on a SolidRun Honeycomb built around NXP’s Layerscape LX2160A processor featuring 16 Arm Cortex-A72 cores. The system is equipped with 32GB of memory and running the Arm version of Ubuntu 22.04. The dependencies of state-of-the-art MCU fuzzers (e.g., AFL-Unicorn) require that these are evaluated on machines powered by CPUs that implement the Intel x86-64 ISA. Thus, we use machines equipped with a 16-core Intel Xeon Gold 5218 processor (hyperthreading disabled) and 64GB of RAM for their experiments. These machines also run Ubuntu 22.04. Furthermore, due to the randomness inherent in fuzzing, we conduct ten 24-hour trials of each fuzzing campaign and report average, minimum and maximum numbers where appropriate, following established guidelines [44]. For a fair evaluation, we kept the fuzzing parameters identical across all the campaigns. Each fuzzing campaign was configured with the same timeout, compute resources and bootstrapped with the same set of seeds. We conducted all experiments using our set of minimal input seeds, minimizing the risk of impacting coverage increases due to seeds crafted with firmware-specific knowledge. We did not reuse seeds from P2IM or Fuzzware. Instead, our initial minimal seed consists of the three-byte string `foo`. Seeds from related work are either hand-crafted to reach complex firmware functionality or large undirected seeds (e.g., configuration files for non-related software). Starting with non-minimal seeds hinders comparability of the fuzzing campaigns, because the fuzzers process the inputs differently depending on their peripheral modeling strategy. The start seeds might therefore allow one fuzzer to already bypass a difficult condition that the other fuzzers get stuck on due to the different parsing of the seeds.

6.2 Fuzzing Throughput

We measured the throughput (i.e., the number of executions per second) of EmbedFuzz compared to Fuzzware and P2IM. Throughput is an important factor for fuzzing since exploring more inputs directly correlates with a larger probability of discovering bugs. EmbedFuzz improves fuzzing throughput by a factor of up to 8x in comparison to its competitors, as shown in Figure 2.

As a notable exception, EmbedFuzz and Fuzzware both exhibit lower executions per second on the Reflow Oven firmware than P2IM. This observation can be explained through the excessive use of time measurements in the firmware during which the firmware loops until an internal counter incremented on each SysTick interrupt reaches a certain value. We note that our virtual clock fast-forwarding design as described in Section 4.3 fast-forwards virtual time until the next interrupt is triggered. Our mechanism cannot account for internal counters managed by the firmware. Our instruction counter based virtual clock in the configuration used for our evaluation increments this counter slower than the implementations of Fuzzware and P2IM.

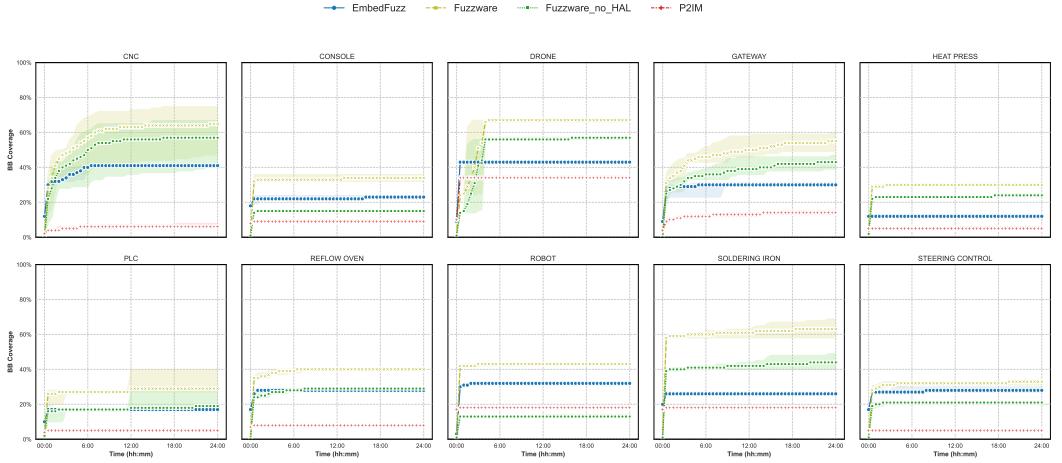


Fig. 3. Code coverage over time for real-world firmware binaries across ten 24 hour trials. The median percentage of uniquely discovered basic blocks, as well as the minima and maxima for each trial are illustrated.

EmbedFuzz shows similar throughput to Fuzzware on the Soldering Iron firmware based on FreeRTOS [2]. The complexity of FreeRTOS as further described in Section 6.7 requires fallbacks to our trap-and-emulate approach, incurring overheads through the context switches in and out of the host OS kernel during trap dispatching.

Based on our experiments, approximately 65% of the handlers in MCU firmware are trivial low-fidelity handlers that simply return a constant, such as the ones used for hardware initialization functions. As an example, configuring the SysTick timer in the STM32 HAL incurs calls to multiple functions for configuring system clocks, choosing the clock to be used as a source for the SysTick, retrieving the clock's frequency, and finally configuring the SysTick itself. Only the last function in this call sequence is of interest for our HLM approach, since this is the only function that we concretely use for enabling the emulated SysTick timer. The previous functions therefore can without loss of functionality be replaced with simple native handlers that return a value indicating successful execution of the requested operation, speeding up execution.

Additionally, we compare the average number of executions per second of EmbedFuzz running natively on Arm and on x86_64 emulated with qemu-user. This experiment highlights the performance improvements through our transplantation approach, resulting in an on average seven times higher fuzzing throughput.

Based on the results of our experiments, we conclude that EmbedFuzz exhibits higher throughput in executions/sec during fuzzing campaigns than previous re-hosting systems, answering RQ 1.

6.3 Coverage Over Time

If a fuzzer covers more execution paths in the Program Under Test (PUT) faster, it can earlier probe the input space along these paths. Thus, exploration is a prerequisite for finding bugs. In our experiments, we measure the number of uniquely discovered basic blocks by EmbedFuzz and compare against P2IM and Fuzzware over ten 24-hour runs.

Figure 3 summarizes our experiments on the number of discovered basic blocks over time. Given that Fuzzware and P2IM can reach many more basic blocks than EmbedFuzz (e.g., basic blocks beneath the HAL layer), it is surprising to see that EmbedFuzz outperforms P2IM in all ten cases.

Table 2. Bugs in our evaluation data set triggered by the three fuzzers. Bug ID refers to the identifier assigned by Fuzzware’s experiments available at <https://github.com/fuzzware-fuzzer/fuzzware-experiments>. Abbreviations: **E**MBED**F**UZZ; **F**UZZ**W**ARE. Symbols: ✓ == bug triggered, ✗ == bug not reproducible, ? == false positive bug

Bug ID	Target	Description	EF	FW	P2IM
11	CNC	Stack OOB write	✓	✓	
12	Gateway	OOB write in HAL	✓	✓	✓
13	Heat Press	Buffer overflow		✓	✓
14	PLC	Missing bounds check	✓	✓	✓
15	PLC	Missing bounds check	✓	✓	
16	PLC	Missing bounds check	✓	✓	✓
17	PLC	Missing bounds check	✓	✓	✓
18	CNC	CNC input validation		✗	
20	Robot	Initialization race		?	
24	Gateway	Missing pointer check	✓	✓	✓
25	PLC	Missing initialization		?	
26	Reflow Oven	Missing pointer check		?	
Total			7	8	6

For most of the firmware, Fuzzware covers significantly more basic blocks than EmbedFuzz and P2IM. Analysis of seed-generating traces shows that the majority of these basic blocks are located in the HAL libraries of the firmware. However, as EmbedFuzz intercepts and redirects control at the entry of HAL APIs, the HAL function implementations become unreachable code for our approach. We statically analyze the firmware binaries to determine basic blocks only reachable from the HAL functions EmbedFuzz intercepts. We then evaluate the coverage attained by Fuzzware when subtracting HAL code as determined in our static analysis. This approach provides a lower bound on the excluded basic blocks, since targets of function pointers calculated at runtime cannot be reliably detected. As shown in Figure 3, EmbedFuzz covers on par or more of the firmware’s code above the HAL than Fuzzware in five out of ten cases.

For Console, Heat Press, and Robot, all fuzzers show signs of early coverage stagnation. A manual investigation revealed that there are conditional guards in the firmware that are difficult to be satisfied with brute-force guessing. Such guards are well-known to prevent fuzzers from reaching and covering deeper parts of the code. However, improving fuzzer capability to address challenges such as magic values and checksums is outside the scope of this paper and has been the focus of [13, 56, 64].

In summary, EmbedFuzz follows a high-level modeling approach and focuses on the task logic above the HAL of an embedded firmware. Our coverage experiments show that EmbedFuzz can outperform P2IM’s and Fuzzware’s coverage achieved during the fuzzing campaigns for our dataset, positively answering RQ 2.

6.4 Bug Finding and Debugging Capabilities

We investigated the crashes generated by EmbedFuzz, Fuzzware, and P2IM. Specifically, we first classified the crashing test cases generated across experiments based on stack trace analysis. Thereafter, we conducted a manual root-cause analysis for each group to determine the type and severity of each bug. The flexibility of EmbedFuzz in debugging helped us to readily perform crash triaging. For this use case, transplantation enabled us to utilize fully-fledged debugging tools such

as GDB. These tools are suitable for debugging native userspace applications without going through debugging stubs in emulators or hardware debuggers.

In 24 hours of fuzzing of each firmware, EmbedFuzz, Fuzzware, and P2IM reported seven, eight, and six distinct bugs, respectively. We provide details on the triggered bugs in [Table 2](#). Note that we focus on the subset of bugs reported by Fuzzware that are triggered above the HAL and that are hence reachable by EmbedFuzz. However, the bug with ID 18 in Fuzzware’s experiments was not reproducible. The bugs with IDs 20, 25 and 26 are false positives only triggered due to Fuzzware’s interrupt modeling. These bugs occur when an interrupt is triggered before it has been configured in the firmware. Fuzzware triggers available interrupts from the start of the execution onwards. EmbedFuzz on the other hand only starts to trigger an interrupt once it has been configured, which EmbedFuzz detects by hooking the corresponding HAL APIs.

Our analysis of the encountered bugs reveals oftentimes complex path constraints to trigger a bug. A symbolic execution-based fuzzer such as Fuzzware may overcome such constraints more easily than other approaches thanks to its additional analyses. We argue that this observation showcases the advantages and disadvantages of different peripheral modeling approaches rather than transplantation.

In summary, EmbedFuzz was able to trigger seven bugs reported by Fuzzware, and outperformed P2IM in its bug-finding capabilities, answering [RQ 3](#). EmbedFuzz’s transplantation approach significantly simplified crash triaging thanks to the availability of debugging tools for a normal user space process instead of emulator-introduced debugging stubs.

6.5 Power Efficiency

Power efficiency is crucial to modern datacenter design and must be taken into account for large-scale fuzzing campaigns. NXP’s LX2160A CPU offers 16 cores with a Thermal Design Power (TDP) below 30W [30, 54], while Intel’s Xeon Gold 5128 offers 16 cores with a TDP of 125W [36]. Since the architectures and technical specifications of those CPUs as detailed in [Table 3](#) are fundamentally different, a power efficiency argument cannot be made on the TDP alone. For this reason, we measure the actual energy consumption of the full systems running the fuzzing campaigns. We encounter total system power draws of 47.0W and 197.1W on average for the Arm and Intel x86_64-based servers, respectively. Our observations match the theoretical difference in TDP, with an over four-fold increase in power consumption for the Intel Xeon-based server in comparison to NXP’s LX2160A CPU.

Considering the LX2160A’s lower operating frequency, the lower core count if considering Simultaneous Multithreading (SMT), and less cache memory available to the CPU, NXP’s LX2160A CPU at first glance seems to be less powerful than the Intel Xeon Gold 5128 CPU we use for the evaluation of EmbedFuzz against previous work. However, our numbers from [Section 6.2](#) show greatly increased throughput when executing firmware natively with EmbedFuzz instead of relying on emulation. In our evaluation, EmbedFuzz therefore runs fuzzing campaigns with a 4-16 increased power efficiency depending on the analysis target in comparison to emulation-based systems. This observation holds true even when generously assuming that enabling SMT would double the processing power available to the Intel CPUs without increasing power consumption.

As a result, transplantation instead of emulation-based re-hosting has a significant impact on the energy consumption of large-scale fuzzing campaigns, while still increasing the campaigns’ performance. Lower energy consumption implies simpler and cheaper infrastructure due to lower requirements, for example on cooling or power supplies. The reduced cost for electrical power itself must also be taken into consideration. Consequently, EmbedFuzz significantly reduces the required computational resources for fuzzing campaigns, answering [RQ 4](#).

Table 3. Comparison of the NXP LX2160A [30, 54] and Intel Xeon Gold 5128 [36] CPUs. Power draw numbers refer to the full system’s wall power draw, measured over 24 hours during fuzzing campaigns.

	LX2160A	Xeon Gold 5128
Architecture	Armv8-A	x86_64
Cores	16	16 (+16 with SMT)
Base frequency	2.2GHz	2.3GHz
Turbo frequency	2.2GHz	3.9GHz
Cache (L1 + L2 + L3)	18MB	22MB
TDP	30W	125W
Power draw (max)	47.1W	200.0W
Power draw (avg)	47.0W	197.1W
Power draw (idle)	29.9W	84.0W

6.6 C/Python Peripheral Handlers

EmbedFuzz supports peripheral handlers implemented both in C and Python. C handlers in our implementation are required to follow the corresponding HAL function’s prototype. Arm’s ABI, the AAPCS, then ensures correct propagation of arguments and return values as well as saving and restoring of callee-saved registers. Thanks to an embedded Python interpreter in the runtime, EmbedFuzz also permits re-using HALucinator’s peripheral handlers with minimal to no changes. This feature allows for less engineering work to be conducted by a developer and rapid prototyping thanks to the simplicity of the Python language.

However, natively compiled handlers provide significant performance advantages over Python handlers, which is an important factor for high-performance dynamic analysis. In our experiments, fuzzing campaigns leveraging C handlers ran with up to 17x improved executions/sec in comparison to their Python handler based counterparts, with an average increase of 5x. We attribute the slowdown incurred to the interpreted nature of the Python programming language. Additionally, we observed the Python interpreter to issue a multitude of system calls for each imported Python module, both for locating and actually interpreting the module. These system calls, causing context switches into and out of the OS kernel, incur additional overhead.

Based on our experiments, we conclude that pre-existing Python handlers taken from HALucinator provide an excellent base for quick prototyping, but that peripheral handlers should be implemented in C to fully reap the benefits of EmbedFuzz’s native execution.

6.7 Case Study: The Soldering Iron Firmware

The Soldering Iron firmware from the P2IM dataset used for our evaluation is based on IronOS [37]. This firmware is available for a variety of small, handheld soldering irons and is built on top of FreeRTOS [2]. The RTOS runs multiple tasks, each with a dedicated responsibility such as temperature sensing, power control, or handling UI tasks via builtin buttons and screens. The task scheduling functionality of an RTOS such as FreeRTOS highlights the importance of accurate system modeling in a re-hosting environment as provided by EmbedFuzz’s transplantation approach.

First, the firmware executes an svc (supervisor call) instruction to call into FreeRTOS’s task scheduler once the system is initialized. As described in Section 5, EmbedFuzz substitutes the svc instruction in the firmware binary with a trap to our runtime in which we execute code mimicking the expected behavior. Hal-fuzz, based on the Unicorn emulation engine [58], does not install hooks in the ISA emulation engine for handling this instruction, effectively ignoring its effects. Safirefuzz also neglects to handle this instruction but causes side effects on the host system due to native

execution of the instruction. On an Arm-based Linux system, a supervisor call is the equivalent to x86’s `syscall` instruction, calling into the kernel and executing unintended system calls.

Second, once an interrupt is triggered on Cortex-M, the CPU pushes execution context onto the stack, executes the interrupt handler as defined in the vector table, and finally pops execution context from the stack before continuing execution of user code. FreeRTOS leverages this hardware-supported context save and restore around ISRs to implement task switches by setting the stack for each task up accordingly. Once the initial code triggers a software interrupt via the `svc` instruction, the corresponding interrupt handler only needs to switch the stack pointer to point to another task’s stack. Then, the ISR simply returns, after which the CPU restores the new task’s context from the stack. EmbedFuzz supports this behavior in software, pushing and popping execution context to and from the stack around the ISR in a few assembly instructions. Hal-fuzz also implements these exception entry and return routines in software, but needs to interact with the Unicorn ISA emulator to modify its register and memory state. Safirefuzz lacks an implementation for replicating this hardware behavior.

Finally, FreeRTOS tasks leverage the PendSV interrupt to give up control and let the scheduler switch to the next task. While the interrupt triggered by an `svc` instruction is synchronous, i.e., immediately triggered, the PendSV interrupt is asynchronous. The firmware sets a flag in a system control register to mark the interrupt as pending and it is triggered once no interrupts with a higher priority need to be served anymore. EmbedFuzz adds instrumentation to the trampolines used for fuzzing coverage collection and virtual clock emulation, which checks whether this flag is set. If it is, the same context save and restore routines as described above are executed around the ISR for the PendSV interrupt. Both hal-fuzz and Safirefuzz are do not replicate this behavior of a real MCU.

While this case study focuses on a FreeRTOS-based firmware, other RTOSs such as RIOT used in the Console firmware in our dataset follow the same model for task switching. In consequence, both hal-fuzz and Safirefuzz are incompatible with the P2IM firmware data set due to shortcomings in modeling the MCU’s behavior.

In response to [RQ 5](#), the fidelity of transplantation for high-performance dynamic analysis of embedded firmware, we argue that EmbedFuzz’s prototype is the only approach providing both high execution speeds based on native execution and semantically correct execution through transplantation.

6.8 Scalability

Transplantation generally does not require manual engineering efforts from an analyst. The required manual tasks for peripheral modeling and harnessing in our EmbedFuzz prototype solely stem from HALucinator’s HLM approach. In this approach, modeling a peripheral is an initial one-time manual effort which amortizes with each additional firmware that uses a modeled peripheral. In our data set, seven out of ten firmware leverage STMicroelectronics’ STM32 HAL, highlighting the possibility to reuse peripheral handlers across firmware. We implement peripheral handlers in C, following the original HAL functions’ interfaces. The implementation of the 49 handlers required to support the firmware in our data set resulted in ~ 1,000 lines of C code.

As described in [Section 5](#), EmbedFuzz also supports prototyping of peripheral handlers in Python based on HALucinator’s interface. In addition to our C handlers, we ported peripheral handlers published with HALucinator to showcase EmbedFuzz’s versatility. Our usage of in-process handlers instead of QEMU for HAL function interception and our custom interrupt-handling mechanism required straightforward modifications to some HALucinator handlers which took one day of engineering effort.

Implementing handlers in C or Python requires minimal effort in the order of a few hours per HAL from an analyst familiar with the HAL. We consequently conclude that the one-time effort

required for peripheral modeling allows an analyst to easily scale EmbedFuzz to other firmware, answering RQ 6.

7 DISCUSSION

Even though EmbedFuzz advances the state of the art based on its contributions highlighted in the previous sections, there are some limitations to the current implementation. We now discuss these limitations, their effect on the general applicability of our system, and how those limitations can be addressed in future work.

High-level Modeling. EmbedFuzz’s reliance on HLM based on the approach provided by HALucinator [17] comes with similar limitations to those presented by Clements et al.. Notably, the firmware to be transplanted needs to be implemented using HAL libraries or embedded OSs providing a similar interface in order for EmbedFuzz to be able to intercept peripheral accesses at this interface. We expect this to be the case for real-world firmware. Without this prerequisite, EmbedFuzz cannot handle peripheral accesses triggered by the targeted firmware.

EmbedFuzz does not require source code access for the full MCU firmware. However, access to the HAL function source code is required to match symbols in the firmware binary to the corresponding function to correctly implement our HLM handler as described in Section 5. If symbols are not available, we use HALucinator’s LibMatch approach which correlates compiled HAL functions with the provided binary to find the correct offsets. Both approaches require access to the HAL function source.

Even though we encounter those limitations, we argue that handling peripheral accesses on the Hardware Abstraction Layer reduces analysis effort due to the reusability of peripheral models across firmware if they rely on the same HAL library. With the HAL libraries being provided openly by MCU vendors to developers, reuse of HAL libraries is widespread and EmbedFuzz therefore greatly benefits from the reusability of its peripheral models.

Transplantation and Native Code Execution. One of the features of EmbedFuzz’s transplantation approach is the performance improvement through native code execution instead of emulation. As outlined before, this requires a CPU supporting the target firmware’s ISA. In our implementation, we run code targeting Arm Cortex-M MCUs on Arm Cortex-A CPUs. EmbedFuzz’s implementation, by design, is also able to execute code targeting Arm Cortex-R based systems as long as the firmware does not leverage an MMU. In this case, the firmware’s physical address space can be modeled in our user space process’s virtual address space without additional address translation costs, as described in Section 4.3. While this approach can certainly be extended to other architectures such as PowerPC featured in MCUs (e.g., NXP’s MPC5xxx MCUs [55]) as well as in high-performance server CPUs (e.g., IBM’s Power10 CPUs [35]), it is not applicable to all MCU architectures. To the best of our knowledge, there are no high-performance CPUs on the market supporting the AVR or XTensa ISAs which excludes MCU firmware based on these architectures from the list of possible transplantation targets.

Transplantation is successful because we leverage a high-performance host CPU feature superset of the low-power MCU features. If a firmware makes extensive use of features not available in Arm Cortex-A CPUs such as a MPU, EmbedFuzz currently does not support transplantation of such firmware. Our prototype can be extended to emulate such behavior by inducing significant performance costs.

Another limitation is the handling of interrupts triggered by peripherals. As described in Section 4.3, we simulate interrupts occurring at regular intervals. We relate the time elapsed to the number of executed instructions. EmbedFuzz falls back to a runtime trap wherever this is not possible, e.g., for software interrupts triggered via an svc instruction. Given peripheral interception,

Table 4. A comparison between EmbedFuzz and a selection of state-of-the-art fuzzing approaches proposed for MCU firmware. Abbreviations: **E**mulated; **N**ative; **H**igh-level (function-level); **L**ow-level (MMIO/register-level); **P**assthrough; **M**MMIO; **D**MA; **H**ard**W**are; **S**ource **C**ode

Objectives	HALucinator [17]	P2IM [23]	Para-rehosting [46]	Fuzzware [60]	μ EMU [76]	Avatar ² [52]	Safirefuzz [62]	EmbedFuzz
Code execution	E	E	N	E	E	E	N	N
Peripheral modeling	H	L	H	L	L	P	H	H
Supported I/O	M, D	M	M, D	M	M	M	M, D	M, D
HW independence	✓	✓	✓	✓	✓	✗	✓	✓
SC independence	✓	✓	✗	✓	✓	✓	✓	✓

this may not accurately match the time executed in real firmware. This is an issue of re-hosting fidelity [72]. As a result, EmbedFuzz’s interrupt simulation is considered a best-effort approach that cannot simulate a real interrupt-based system’s behavior with 100% accuracy. We nevertheless contend that our strategy is more than sufficient for our needs.

8 RELATED WORK

With the wide adoption of fuzzing, a large body of related work has contributed to applying fuzzing to embedded systems to enhance their security.

We analyze existing work targeting embedded system fuzzing along several dimensions: (1) code execution, (2) peripheral modeling approach, (3) supported I/O methods, (4) dependency on hardware, (5) fault detection mechanism, and (6) dependency on source code. Table 4 summarizes a selection of related work and tools, representing all the different categories of approaches laid out in the following.

Hardware-in-the-loop solutions [15, 18, 29, 40, 41, 45, 52, 68, 73] rely on the actual MCU hardware. Such solutions are inherently limited in performance due to communication overhead with the MCU through debug probes and provide poor scalability due to additional hardware costs and configuration efforts. EmbedFuzz overcomes those limitations thanks to peripheral modeling in software and native code execution in a Linux user space process.

Some re-hosting approaches propose modeling peripherals in software at the kernel level, providing data through generic POSIX interfaces to a userspace process [14, 19, 38, 43, 75]. In contrast to EmbedFuzz, these approaches are not applicable to MCU firmware due to their focus on Linux-based embedded firmware.

Symbolically modeling values read from peripheral registers [12, 33, 60, 63, 76] enables fully automated MCU firmware re-hosting. However, as stated by Fasano et al. [22], this technique often over-approximates peripheral capabilities even for simple firmware binaries. We argue that EmbedFuzz’s manual handler implementations provide higher accuracy for possible values returned from peripherals and do not require to run an analysis phase for each firmware to determine an appropriate peripheral model. Modeling peripheral registers by heuristically matching MMIO access patterns to common static register types [23, 32, 51] requires running costly analyses on each new firmware image similar to symbolic modeling approaches.

Alternatively, other approaches [16, 17, 48] model the peripheral behaviors at higher abstraction layers. They leverage HAL library functions for re-hosting and analyzing MCU firmware binaries by intercepting such high-level functions instead of low-level MMIO or register accesses. These approaches leverage ISA translation in an emulator to execute target code, which incurs high runtime overhead and low fuzzing throughput. In contrast, EmbedFuzz executes Cortex-M firmware on Cortex-A processors at native speed using transplantation.

In concurrent work, Safrefuzz [62] (to be published in USENIX Security ’23) also leverages the similarity of the Arm Cortex-M and Cortex-A ISAs, thereby achieving performance gains in fuzzing campaigns. In contrast to EmbedFuzz, Safrefuzz fails to address semantic differences in the ISAs, threatening the semantical correctness of executed code. While Safrefuzz’s dynamic binary rewriting approach at runtime is targeted purely at fuzzing, EmbedFuzz’s combination of static binary rewriting and a minimal runtime generically transplants firmware binaries into a Linux user space process for any dynamic analysis use case.

Source code based approaches such as para-rehosting [46] employ manually implemented peripheral handlers directly compiled into the final binary. This approach relies on firmware source code, which is typically not provided for Commercial-off-the-shelf (COTS) IoT devices. In contrast, EmbedFuzz works on binary firmware without introducing dependencies on the source code.

9 CONCLUSION

EmbedFuzz is the first end-to-end dynamic analysis framework for MCU firmware capable of running its targets at native speed. We implemented EmbedFuzz following our concept of *firmware transplantation*, a novel technique for re-hosting and a faster alternative to previous emulation-based approaches. Firmware transplantation exploits the ISA compatibility of low-end and high-end platforms such as Arm Cortex-M and Arm Cortex-A, respectively. Our system adapts and extends High-level Modeling (HLM) to allow for fuzzing the firmware without real hardware. For our prototype, we employ AFL++ as a drop-in replacement for a modern general-purpose fuzzer. Our evaluation results show that EmbedFuzz outperforms the state-of-the-art in terms of performance by a factor of up to eight, resulting in efficiency gains by a factor of at least four. We highlight the importance of semantically correct execution of firmware code, which EmbedFuzz provides through transplantation. Finally, our bug triaging case studies show that EmbedFuzz facilitates the analysis of bug root causes by enabling existing OS-provided introspection capabilities.

REFERENCES

- [1] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. 2020. μ RAI: Securing Embedded Systems with Return Address Integrity. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. <https://doi.org/10.14722/ndss.2020.24016>
- [2] Amazon Web Services, Inc. [n. d.]. FreeRTOS. <https://www.freertos.org/> Accessed: July 2022.
- [3] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, Monterey California, 311–327. <https://doi.org/10.1145/2815400.2815420>
- [4] Ampere Computing. 2022. Ampere® Altra®. <https://amperecomputing.com/processors/ampere-altra/>
- [5] Apple Inc. 2022. Mac – Apple. <https://www.apple.com/mac/> Accessed: July 2022.
- [6] Arm Limited. 2021. *Armv7-M Architecture Reference Manual*. Technical Report. <https://developer.arm.com/documentation/ddi0403/ee/?lang=en>
- [7] Arm Limited. 2022. *Arm Architecture Reference Manual for A-profile Architecture*. Technical Report. <https://developer.arm.com/documentation/ddi0487/ia/?lang=en>
- [8] Arm Limited. 2022. Mbed – Rapid IoT Device Development. <https://os.mbed.com/> Accessed: January 2022.
- [9] Arm Limited. 2022. *Procedure Call Standard for the Arm® Architecture*. Technical Report. <https://github.com/ARM-software/abi-aa>
- [10] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC ’05)*. USENIX Association, USA, 41.

- [11] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2022. Morphuzz: Bending (Input) Space to Fuzz Virtual Devices. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 1221–1238. <https://www.usenix.org/conference/usenixsecurity22/presentation/bulekov>
- [12] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-Agnostic Firmware Execution Is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference*. ACM, Austin USA, 746–759. <https://doi.org/10.1145/3427228.3427280>
- [13] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.
- [14] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. <https://doi.org/10.14722/ndss.2016.23415>
- [15] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTfuzzer: Discovering Memory Corruptions in IoT through App-Based Fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_01A-1_Chen_paper.pdf
- [16] Abraham A. Clements, Logan Carpenter, William A. Moeglein, and Christopher Wright. 2021. Is Your Firmware Real or Re-Hosted? A Case Study in Re-Hosting VxWorks Control System Firmware. In *Proceedings 2021 Workshop on Binary Analysis Research*. Internet Society, Virtual. <https://doi.org/10.14722/bar.2021.23006>
- [17] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Matthias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1201–1218. <https://www.usenix.org/conference/usenixsecurity20/presentation/clements>
- [18] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 309–326. <https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani>
- [19] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 437–448. <https://doi.org/10.1145/2897845.2897900>
- [20] Digi-Key. 2013. MCUs in Industrial Automation. <https://www.digikey.com/en/articles/mcus-in-industrial-automation>
- [21] Electronics Sourcing. 2017. Reversal of Fortune for Chip Buyers: Average Prices for Microcontrollers Will Rise. <https://electronics-sourcing.com/2017/05/09/reversal-fortune-chip-buyers-average-prices-microcontrollers-will-rise/> Accessed: January 2022.
- [22] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. 2021. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ACM, Virtual Event Hong Kong, 687–701. <https://doi.org/10.1145/3433210.3453093>
- [23] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-Independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1237–1254. <https://www.usenix.org/conference/usenixsecurity20/presentation/feng>
- [24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. <https://www.usenix.org/conference/woot20/presentation/fioraldi>
- [25] November Five. 2017. Withings Body Cardio Teardown. <https://www.ifixit.com/Teardown/Withings+Body+Cardio+Teardown/74987> Accessed: January 2022.
- [26] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. 2021. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event Republic of Korea, 366–378. <https://doi.org/10.1145/3460120.3484748>
- [27] Ghidra Contributors. 2022. Ghidra Software Reverse Engineering Framework. National Security Agency. <https://github.com/NationalSecurityAgency/ghidra>

- [28] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (Jan. 2020), 70–76. <https://doi.org/10.1145/3363824>
- [29] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware Through Automated Re-Hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 135–150. <https://www.usenix.org/conference/raid2019/presentation/gustafson>
- [30] Tom R. Halfhill. 2017. LX2160A Is NXP’s Biggest Multicore. (Oct. 2017). <https://www.nxp.com/docs/en/supporting-information/LX2160A-NXP-Biggest-Multicore.pdf> Accessed: January 2022.
- [31] Jason Harris. 2016. Grbl STM32F4. https://github.com/deadsy/grbl_stm32f4 Accessed: January 2022.
- [32] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, Srdjan Capkun and Franziska Roesner (Eds.)*. USENIX Association, 789–806. <https://www.usenix.org/conference/usenixsecurity20/presentation/harrison>
- [33] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R. B. Butler. 2017. FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2245–2262. <https://doi.org/10.1145/3133956.3134050>
- [34] Marc Heuse, Heiko Eißfeldt, Andrea Fioraldi, Dominik Maier, and Jana Aydinbas. 2022. The AFL++ Fuzzing Framework. <https://aflplus.plus/> Accessed: July 2022.
- [35] IBM. 2022. IBM Power10: Engineered for Agility. <https://www.ibm.com/it-infrastructure/power/power10> Accessed: January 2022.
- [36] Intel Corporation. 2017. Intel® Xeon® Gold 5218 Processor Product Specification. <https://ark.intel.com/content/www/us/en/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html> Accessed: January 2022.
- [37] IronOS contributors. 2023. IronOS – Flexible Soldering Iron Control Firmware. <https://github.com/Ralim/IronOS/>
- [38] Muhui Jiang, Lin Ma, Yajin Zhou, Qiang Liu, Cen Zhang, Zhi Wang, Xiapu Luo, Lei Wu, and Kui Ren. 2021. ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event Republic of Korea, 734–748. <https://doi.org/10.1145/3460120.3484753>
- [39] Ashu Joshi. 2013. Philips Hue: Setup and Teardown. <https://allthingscc.wordpress.com/2013/01/21/phillips-hue-setup-and-teardown/> Accessed: January 2022.
- [40] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*.
- [41] Markus Kammerstetter, Christian Platzner, and Wolfgang Kastner. 2014. Prospect: Peripheral Proxying Supported Embedded Code Testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ACM, Kyoto Japan, 329–340. <https://doi.org/10.1145/2590296.2590301>
- [42] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_04B-2_Kim_paper.pdf
- [43] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *ACSAC ’20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 733–745. <https://doi.org/10.1145/3427228.3427294>
- [44] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 2123–2138. <https://doi.org/10.1145/3243734.3243804>
- [45] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling near-Real-Time Dynamic Analyses of Embedded Systems. In *9th USENIX Workshop on Offensive Technologies, WOOT ’15, Washington, DC, USA, August 10-11, 2015*, Aurélien Francillon and Thomas Ptacek (Eds.). USENIX Association. <https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher>
- [46] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. 2021. From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, Virtual. <https://doi.org/10.14722/ndss.2021.24308>

- [47] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. 2023. ViDeZZo: Dependency-aware Virtual Device Fuzzing. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 3228–3245. <https://doi.org/10.1109/SP46215.2023.10179354>
- [48] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. BaseSAFE: Baseband Sanitized Fuzzing through Emulation. In *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Linz, Austria, July 8-10, 2020*, René Mayrhofer and Michael Roland (Eds.). ACM, 122–132. <https://doi.org/10.1145/3395351.3399360>
- [49] Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331. <https://doi.org/10.1109/TSE.2019.2946563> arXiv:1812.00140 Comment: 29 pages, under submission to ACM Computing Surveys (July 2018) - 2018.12.10 update: correct minor mistakes in overview table - 2019.02.16 update: source clean - 2019.04.08: submission to TSE, 21 pages.
- [50] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 337–348. <https://doi.org/10.1145/2150976.2151012>
- [51] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. 2021. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1938–1954. <https://doi.org/10.1109/SP40001.2021.00018>
- [52] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar²: A Multi-Target Orchestration Platform. In *Proceedings 2018 Workshop on Binary Analysis Research*, Vol. 18. Internet Society, San Diego, CA. <https://doi.org/10.14722/bar.2018.23017>
- [53] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '07*. ACM Press, San Diego, California, USA, 89. <https://doi.org/10.1145/1250734.1250746>
- [54] NXP Semiconductors. 2020. NXP Layerscape LX2160A, LX2120A, LX2080A Data Sheet. <https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-lx2160a-lx2120a-lx2080a-processors:LX2160A> Accessed: January 2022.
- [55] NXP Semiconductors. 2022. MPC5xxx Microcontrollers. https://www.nxp.com/products/processors-and-microcontrollers/power-architecture/mpc5xxx-microcontrollers:POWER_ARCH_5XXX Accessed: January 2022.
- [56] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 697–710. <https://doi.org/10.1109/SP.2018.00056>
- [57] Qualcomm Technologies, Inc. 2022. Smartphone Technology | Processor, CPU & GPU Data. <https://www.qualcomm.com/products/application/smartphones> Accessed: July 2022.
- [58] Nguyen Anh Quynh and Dang Hoang Vu. 2015. Unicorn: Next Generation Cpu Emulator Framework. *BlackHat USA* 476 (2015). <https://www.unicorn-engine.org>
- [59] Majid Salehi, Danny Hughes, and Bruno Crispo. 2019. Microguard: Securing Bare-Metal Microcontrollers Against Code-Reuse Attacks. In *2019 IEEE Conference on Dependable and Secure Computing, DSC 2019, Hangzhou, China, November 18-20, 2019*. IEEE, 1–8. <https://doi.org/10.1109/DSC47296.2019.8937667>
- [60] Tobias Scharnowski, Nils Bars, Moritz Schloegl, Eric Gustafsson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. <https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski>
- [61] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing Using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2597–2614. <https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo>
- [62] Lukas Seidel, Dominik Maier, and Marius Muench. 2023. Forming Faster Firmware Fuzzers. In *USENIX Security*. <https://www.usenix.org/conference/usenixsecurity23/presentation/seidel>
- [63] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. <https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware>

- [64] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. <http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf>
- [65] STMicroelectronics. 2022. STM32 Arm Cortex MCUs - 32-Bit Microcontrollers. <https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html>
- [66] STMicroelectronics. 2022. STM32Cube Development Software. <https://www.st.com/en/ecosystems/stm32cube.html> Accessed: July 2022.
- [67] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education.
- [68] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 291–307. <https://www.usenix.org/conference/usenixsecurity18/presentation/talebi>
- [69] Dmitry Vyukov and Andrey Konovalov. 2022. Syzkaller. <https://syzkaller.appspot.com/upstream/fixed> Accessed: July 2022.
- [70] Dmitry Vyukov and Andrey Konovalov. 2022. Syzkaller: An Unsupervised Coverage-Guided Kernel Fuzzer. <https://github.com/google/syzkaller/> Accessed: July 2022.
- [71] Stacy Wegne. 2018. Fitbit Charge 3 Teardown. <https://www.techinsights.com/blog/fitbit-charge-3-teardown> Accessed: January 2022.
- [72] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. 2021. Challenges in Firmware Re-Hosting, Emulation, and Analysis. *ACM Comput. Surv.* 54, 1 (2021), 5:1–5:36. <https://doi.org/10.1145/3423167>
- [73] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *21st Annual Network and Distributed System Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society. <https://www.ndss-symposium.org/ndss2014/avatar-framework-support-dynamic-security-analysis-embedded-systems-firmwares>
- [74] Michal Zalewski. 2017. American Fuzzy Lop (AFL) Fuzzer. <http://lcamtuf.coredump.cx/afl/> Accessed: January 2022.
- [75] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1099–1114. <https://www.usenix.org/conference/usenixsecurity19/presentation/zheng>
- [76] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-Guided Knowledge Inference. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2007–2024. <https://www.usenix.org/conference/usenixsecurity21/presentation/zhou>

A INSTRUCTIONS REWRITTEN BY EMBEDFUZZ

EmbedFuzz needs to handle a small set of instructions exhibiting different behavior in an ArmCortex-A user space process in comparison to an ArmCortex-M bare-metal firmware binary. We categorize these instructions as follows:

Modifying MCU-specific CPU state. The instructions `msr` (move to special register from register) and `mrs` (move to register from special register) copy information to and from model-specific registers, respectively. We rewrite those instructions to either only change arguments with semantic equivalence (e.g., model the Cortex-M Application Program Status Register (APSR) with Cortex-A's Current Program Status Register (CPSR)) or to emulate their behavior in instructions (e.g., copies to/from unused floating point registers for modeling banked stack pointer registers). Where direct replacement is not possible due to complex logic, we replace the instruction with a `bkpt` instruction to trap-and-emulate in our runtime.

Software interrupts. The svc instruction both on Arm Cortex-M and Cortex-A causes a software interrupt to be raised. Whereas the firmware expects the corresponding interrupt handler to be executed, such a software interrupt in a Linux user space process causes a transition to the kernel and execution of a syscall. In order to properly execute the firmware’s interrupt handler and not issue unexpected syscalls on our fuzzing system, we replace svc instructions with bkpt instructions and forward control flow to the interrupt handler from our runtime’s trap handler triggered as a result of the bkpt.

Coprocessor accesses. On Arm MCUs, up to 16 coprocessors can be defined for adding features to the main processor. Any instruction executing on the main CPU is also streamed to a coprocessor and executed on the coprocessor if necessary [6].

Hardware floating point support implemented on a coprocessor does not require special handling by EmbedFuzz, since the corresponding floating point instructions are transparently executed on the floating-point implementation of the host Cortex-A CPU.

In our data set (see [Section 6](#)), we did not encounter custom coprocessors. In case a custom coprocessor is employed, the instructions cdp/cdp2 (coprocessor data processing), mcr/mcr2 (move to coprocessor from Arm register), mcrr/mcrr2 (move to coprocessor from two Arm registers), mrc/mrc2 (move to Arm register from coprocessor), mrrc/mrrc2 (move to two Arm registers from coprocessor), ldc/ldc2 (load coprocessor), and stc/stc2 (store coprocessor) need to be rewritten accordingly. Depending on the coprocessor’s functionality, those instructions can either be rewritten inline with equivalent functionality, or trap into the runtime for corresponding emulation.