SHiFT: Semi-hosted Fuzz Testing for Embedded Applications

Alejandro Mera and Changming Liu, *Northeastern University;* Ruimin Sun, *Florida International University;* Engin Kirda and Long Lu, *Northeastern University*

https://www.usenix.org/conference/usenixsecurity24/presentation/mera

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

SHiFT: Semi-hosted Fuzz Testing for Embedded Applications

Alejandro Mera
Northeastern University

Changming Liu
Northeastern University

Ruimin Sun
Florida International University

Engin Kirda
Northeastern University

Long Lu
Northeastern University

Abstract

Modern microcontrollers (MCU)s are ubiquitous on critical embedded applications in the IoT era. Therefore, securing MCU firmware is fundamental. To analyze MCU firmware security, existing works mostly adopt re-hosting based techniques. These techniques transplant firmware to an engineered platform and require tailored hardware or emulation of different parts of the MCU. As a result, security practitioners have observed low-fidelity, false positives, and reduced compatibility with real and complex hardware.

This paper presents SHiFT, a framework that leverages the industry semihosting philosophy to provide a brand-new method that analyzes firmware natively in MCUs. This novel method provides high fidelity, reduces false positives, and grants compatibility with complex peripherals, asynchronous events, real-time operations, and direct memory access (DMA). We verified compatibility of SHiFT with thirteen popular embedded architectures, and fully evaluated prototypes for ARMv7-M, ARMv8-M and Xtensa architectures. Our evaluation shows that SHiFT can detect a wide range of firmware faults with instrumentation running natively in the MCU. In terms of performance, SHiFT is up to two orders of magnitude faster (i.e., $\times 100$) than software-based emulation, and even comparable to fuzz testing native applications in a workstation. Thanks to SHiFT's unique characteristics, we discovered five previously unknown vulnerabilities, including a zero-day on the popular FreeRTOS kernel, with no false positives. Our prototypes and source code are publicly available at https://github.com/RiS3-Lab/SHiFT.

1 Introduction

Embedded devices are of paramount importance for the modern world. They control critical systems in many domains with real-time responsiveness and low power consumption (e.g., Internet-of-Things, Cyber-Physical Systems, etc.). Microcontrollers, the primary functional component of deeply embedded devices, are becoming increasingly powerful, and rich in network connectivity [20], thus facing threats that pullulate in the cyberspace [40]. MCU security, therefore, is critical to users and the operators of these devices.

In recent years, we have seen a significant rise in attacks targeting embedded and IoT devices [27, 40], where attackers can steal private and critical information of the infected devices, or even manipulate the behavior of the connected physical infrastructure. Such attacks have been studied [68] and reported in smart homes [21], connected vehicles [24, 56], intelligent factories [49], power grids [45], and more.

The attacks usually result from bugs and vulnerabilities rooted in the MCU firmware in production environments. Therefore, detecting and eliminating MCU firmware bugs before deployment is critical because vendors may fail to patch firmware once a vulnerability is known, and is being exploited. A *notable example* is the unfixable firmware bug affecting a DMA buffer in the first generation of the Nintendo Switch. This bug compromises an early boot stage of the console's system on a chip (SoC) and the whole system security [60]. Even when patching after deployment is possible, a complete shutdown and reboot of the physical infrastructure might be needed, hence leading to high financial and operative costs. Thus, firmware developers must find bugs as early as possible at the source code level, leveraging in-house testing [46].

Researchers have explored various approaches to improve the security of embedded devices, including remote attestation [69], compartmentalization [38, 53], and dynamic and static firmware analysis [6, 34, 58]. Notably, dynamic analysis has elicited the interest of security practitioners because it is less prone to producing false positives, adds no runtime overhead, and it is more practical in identifying bugs before deployment. However, contrary to software running in a workstation, firmware is not directly compatible with the majority of traditional security approaches because of various hardware and software constraints of the MCU platform [18, 46, 62].

To solve these limitations and specifically enable dynamic analysis, previous works have explored several *re-hosting* techniques, such as pure software-based emulation [25, 26], hardware-in-the-loop (HiL) [43, 44], and symbolic abstractions [23,30]. Despite these efforts, these solutions are limited in terms of *fidelity*, which describes how well the behavior of an emulation system mirrors its physical counterpart (i.e., the real hardware [33,71]). Low fidelity reflects many issues in firmware analysis, such as limited compatibility with complex peripherals, asynchronous events, real-time operations and DMA. As a result, critical bugs may not be found, or bugs that are reported may not be easy to reproduce nor observed on real hardware (i.e., false positives).

In this work, we propose SHiFT, a fuzz testing framework that introduces "semihosting" as a new re-hosting technique. SHiFT allows the firmware running natively in an MCU to consume fuzzing services in a workstation. SHiFT is novel in the sense that it runs most of the fuzzer's infrastructure and the target firmware entirely in the MCU, without requiring software-based emulation of peripherals, and providing high fidelity.

SHiFT maintains flexibility and practicality in detecting bugs and vulnerabilities early in firmware development. This characteristics benefits first-party developers that have access to source code and functional testing tools, but do not have automated methods to test firmware–from a security perspective–directly on hardware. Consequently, developers can integrate SHiFT in existing development pipelines that embrace continuous integration and continuous delivery (CI/CD) maintaining short development cycles with speed, reliability and security.

SHiFT's implementation involves two components: an MCU fuzzing cluster running FreeRTOS-MPU tailored to support coverage-guided fuzzing and sanitizer instrumentation, and a workstation running a mutation engine. SHiFT's implementation is vendor-agnostic, and portable to many MCU development platforms. We evaluated SHiFT's compatibility with relevant embedded architectures, its fuzzing performance, and fault detection capabilities. The results show that SHiFT is compatible with popular embedded architectures. Its performance is vastly superior to the software-based state-of-the-art emulation, and even comparable to native fuzzing in a workstation. In our work, the unique characteristics of SHiFT allowed the detection of previously-unknown bugs (that we reported responsibly to the vendors) that are easily verifiable in real hardware.

In summary, this work makes the following contributions:

- We propose a novel re-hosting technique based on the *semihosting* philosophy that, contrary to software-based emulation or HIL, executes the firmware under test entirely in an MCU with high fidelity.
- We implement, evaluate and will make public SHiFT prototypes for various popular embedded architectures.
- We demonstrate that by systematically lifting specific limitations of an MCU, it is possible to use MCU devices to build practical security tools, which helped to identify five previously-unknown vulnerabilities.

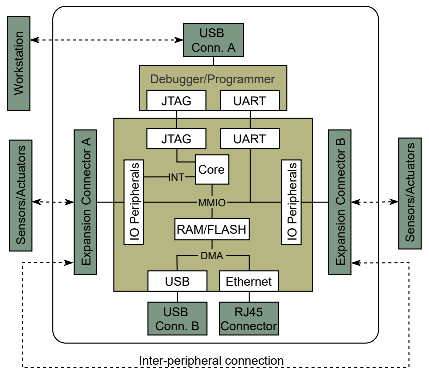

Figure 1: Reference MCU development board architecture. Dashed lines denote external wiring connections, sensors, actuators, and off-chip peripherals added by developers for developing and testing. Adapted from [55].

2 Background

Previous work [59] has classified embedded systems according to the type of OS that they run, and their specific hardware requirements. Our work focuses on devices based on MCUs that implement a specialized RTOS, or a bare-metal design (i.e., Type II, and III, respectively). Devices that have a memory management unit (MMU), and that support a general OS such as Linux (i.e, Type I) are not considered in this work because of their similarity to traditional computers, and the vast availability of security tools.

2.1 The MCU ecosystem

MCUs are tiny self-contained[1] computers that are constrained in computing power, hardware features and data storage. Nevertheless, these devices are unique in undertaking mission-critical tasks, satisfying real-time constraints, and being energy-efficient for battery-powered scenarios [53]. MCUs usually implement a single address space with specific physical addresses mapped to RAM, FLASH and peripherals. The core processor accesses and interacts with these components using three main methods: memory-mapped IO (MMIO), DMA and interrupts (INT) as depicted in Figure 1.

The software executed by an MCU is a monolithic binary (i.e., firmware) that contains the complete software stack including drivers, communication protocols, user applications, and optionally, a scheduler library or RTOS. Firmware is usually developed on workstations equipped with a cross-compiler that generates a binary file. The binary file is then flashed and debugged on a development board. Figure 1 depicts a generic MCU development board that includes a target MCU, expansion connectors to access on-chip peripherals, a terminal connection (USART), and a debugger/programmer.

[1]Processor core, RAM, FLASH and I/O hardware are integrated in a single die.

2.2 Firmware security analysis

Firmware analysis is a challenging endeavor because the security tools and methods—used on standard software and computers—are not directly compatible with firmware and the operative conditions of MCUs. This incompatibility is a consequence of the following embedded devices' characteristics: 1) strong coupling among firmware, sensors and peripherals that requires non-standardized inputs interfaces [34]; 2) diversity of software and hardware that precludes generalization of methods that either rely on specific instruction set architecture (ISA) tooling, or assume the existence of specific software abstractions provided by a common OS [33]; 3) the absence of specialized instrumentation and run-times that can work in bare-metal, or RTOS designs [18,46,59]; and 4) MCU hardware limitations, or minimalistic firmware designs that simply hide crashes and memory corruptions without proper exception management [46,59,62].

Security practitioners have embraced *re-hosting* as the mainstream technique to iron the incompatibility issues of firmware security analysis. "Re-hosting specifies that a binary that would run on a specific hardware is instead run on a host system using system emulation" [71]. Conceptually, re-hosting aims to enable dynamic firmware testing by transplanting firmware from the actual device into an engineered environment that model the original hardware behavior, and executes firmware with visibility and control.

Fasano, et al. [33] classified re-hosting techniques into three non-exclusive categories: *pure software-based emulation*[2] that emulates or replaces all necessary components to run a particular firmware; *HiL* that partially emulates the MCU components and forwards interactions to the actual device when the emulator lacks component models that firmware needs to interact with; and *symbolic abstraction* that replaces interactions between firmware and hardware with symbolic values.

2.3 Open challenges and limitations

Despite the efforts made by the re-hosting community to enable firmware analysis, developers face challenges and limitations when using their SoTA solutions. In the following subsections we systematically describe these issues and how they affect dynamic firmware analysis.

2.3.1 Reduced Fidelity

Re-hosting based on soft-emulation inevitably approximates the behavior of real hardware at its best. This approximation compromises the fidelity of the firmware-hardware interactions and precludes the identification of firmware issues that are closely associated with hardware behavior [51, 76]. A non-exhaustive categorization of these bugs that are hardly observable on soft-emulation platforms includes:

- **Post-silicon and peripheral interfacing bugs:** these bugs are rooted in the silicon implementation [57] or in the integration of an MCU with peripherals due to faulty designs or electrical connections. While impossible to observe on emulators, these issues can compromise firmware execution on the actual device and require vendor-specific firmware "work-arounds" or silicon revisions to be solved [67].

- **Timing and synchronization issues:** this category is related to race conditions, time drifting, time constraint violations, RTOS priority inversions, and wrong handling of interrupts. Current solutions use the number of executed basic blocks as a heuristic to trigger interrupts in a round-robin fashion [34, 63]. This method lacks granularity and does not reflect the concatenation of asynchronous events observed on an actual device.

- **Multi-master violations:** these issues are associated with DMA operations performed by dedicated controllers and peripherals with DMA capabilities. Besides the notable effort made by the authors of DICE [54], most re-hosting works do not support DMA or require non-scalable manual assistance to provide limited support for these operations. Thus, current solutions cannot exercise and analyze several code sections that depend on DMA.

2.3.2 Reduced observability

The fault detection techniques of current re-hosted solutions are based on experimental-only methods [35,50], require considerable engineering effort [26,46], or only provide coarse-grained detection capabilities [17, 34]. This last method is widely adopted and relies on major memory segment violations (i.e., when reading or writing operations occur on reserved memory areas). Unfortunately, these major violations usually do not occur, or they occur later during firmware execution [59]. Potentially observing a fault later during execution is still valuable for firmware analysis, but it demonstrates the re-hosting limitations on detecting specific faults when they occur.

2.3.3 Limited compatibility

This limitation has two dimensions: First, for soft-emulation approaches the current solutions are not compatible with complex peripherals since these solutions can only model simple peripheral-processor interactions [76]; and second, the methods used by re-hosting under the HiL category, with exception of GDBFuzz [31], usually require custom hardware only available for academic use [28,44], or proprietary commercial libraries and expensive debugging dongles [47]. These

[2] In the following sections, we will refer to "pure software-based emulation" as soft-emulation for clarity and brevity.

exclusive requirements make these methods far from being used in practice due to hardware and software unavailability, cost and complexity.

3 System Design

SHiFT addresses the aforementioned challenges based on a novel re-hosting technique that borrows the *semihosting*[3] industry philosophy. Semihosting enables firmware, running natively in an MCU, to use facilities available in a workstation [2]. In the context of dynamic firmware analysis, SHiFT is a fuzz testing framework running on MCUs that uses fuzzing facilities provided by a workstation (i.e., a semi-hosted fuzz testing framework).

3.1 Design assumptions and scope

Our design uses the popular ARM Cortex-M architecture and the FreeRTOS-MPU [7] as a basis. However, the principles and design considerations are non-exclusive, and are applicable to other RTOS, bare-metal designs and architectures as shown in §5.

We assume the tester is a *first-party developer* of embedded devices who has access to a development board, and the source code of the complete firmware, or at least the software components that will be dynamically analyzed. Having access to the development board and source code is a realistic assumption considering that our method is meant to be integrated as part of the development, and in-house testing of an embedded device. It is worth noting that having access to source code does not solve various challenges associated with the MCU constraints and re-hosting that we summarized earlier. Further, previous works [18, 46, 59, 63] have also identified and discussed these remaining challenges, and have agreed with our observation.

We also assume the tester has access to a GCC compiler that supports coverage-guided fuzzing (SANCOV), Address Sanitizer (ASAN) [65], and optionally Undefined Behavior Sanitizer (UBSAN) instrumentation. We assume the run-time of the instrumentation is not available. These assumptions are reasonable and realistic because GCC mainstream supports ASAN and UBSAN since version 4.8 and SANCOV since version 6.3 [5, 8]. Further, the instrumentation run-time is OS-specific, and unavailable for Type II and III devices [18, 46]. We validate our assumptions and extend the compatibility analysis of our solution in §5.

3.2 SHiFT overview and workflow

SHiFT is delivered as a framework that includes ready-to-use components to implement a semihosted fuzzer in an MCU.

[3]Originally termed by ARM [15], semihosting has been used in embedded devices for decades [12].

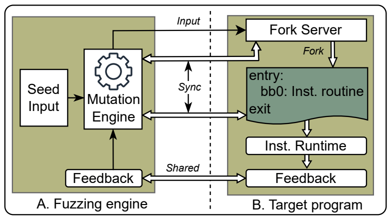

Figure 2: Canonical fuzzer. A. and B. are processes of the same host.

This framework is accompanied by exemplary templates (i.e., projects that integrate all SHiFT's components in a compilable firmware), a tailored compiler, and a proxy running in a workstation.

SHiFT's workflow can be summarized in four steps done by the tester: integration, configuration, compilation, and fuzz testing.

Integration: the tester combines all the source code to build a semihosted fuzzer and a target application in a single binary. For a given firmware codebase, the tester can choose between porting the complete target firmware to any of the exemplary templates, adding the SHiFT framework components to the existing firmware codebase, or porting only specific functions or libraries to the exemplary templates.

Configuration: the tester configures SHiFT components to deliver the fuzzing input to the target application according to the integration method and the portions of the target application to be tested.

Compilation: the tester adds the corresponding compilation flags to instrument the files that will be tested and compile the firmware into a single binary using our tailored compiler.

Fuzz testing: the tester flashes the binary into the development board, or a cluster of them, and launches the fuzzing campaign using the proxy provided by our framework.

3.3 Decoupling a canonical fuzzer

A canonical fuzzer contains a fuzzing engine and a target program running in the same host as mutually-isolated processes (A and B in Figure 2, respectively). The fuzzing engine uses communication primitives (e.g., pipes or shared memory) to synchronize execution, feed input, and retrieve feedback from the target. The target program leverages OS facilities (e.g., syscalls, vitual memory, and error handling), compiler instrumentation, and runtimes to fork, execute, and record feedback of instrumented routines. We call collectively these OS facilities the *fuzzer's infrastructure*.

SHiFT instantiates a semi-hosted fuzzer design by decoupling a canonical fuzzer, and defining two major components: a fuzzing workstation, and an MCU fuzzing cluster (A and B in Figure 3, respectively). In the following sections, we describe our design considerations for each of these major components, and describe how we tackle the unique challenges that the MCU platform pose in terms of constrained

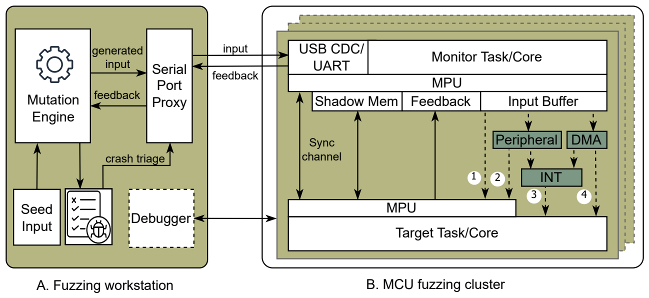

Figure 3: SHiFT framework architecture. Dashed line denotes optional component not needed during the fuzzing campaign.

computing power and memory, lack of OS and hardware facilities, and lack of specialized sanitizer runtimes.

3.4 The fuzzing workstation

We designed the workstation to generate the fuzzing input because it has enough computing power and storage capacity to execute SoTA genetic algorithms and mutation strategies. Thus, the fuzzing workstation integrates a classic feedback-based fuzzing engine as a drop-in component. It also includes a proxy that mimics the target program of the canonical fuzzer. This proxy abstracts the same communication primitives of the original target. However, the proxy merely delivers input, and awaits feedback. It does not control any of the aspects of the target firmware execution and fuzzer's infrastructure.

This independence of the firmware execution has two consequences: first, it instantiates the semihosting philosophy by implementing the fuzzer's infrastructure in the MCU and consuming the fuzzing input from the workstation; and second, it drastically reduces the communication bandwidth between the workstation and the MCU cluster. This last point is a significant advantage of our design compared with classic HiL approaches. HiL, such as Avatar [58] or uAFL [47], uses a debugging dongle to control the MCU from the workstation to mimic the functionality of the fuzzer's infrastructure. This synchronization is complex and requires huge bandwidth, which is mitigated by proprietary debugging dongles. Nonetheless, it reduces the throughput of the fuzzing campaign [59].

3.5 The MCU fuzzing cluster

The MCU fuzzing cluster component implements a fully-functional fuzzer (i.e., the fuzzer's infrastructure) except for the input generator, which we deemed better fits the characteristics of a workstation, as we described earlier. The fuzzing cluster consumes the mutated input from the fuzzing workstation, controls and executes the target program (i.e., firmware), collects coverage, manages errors or faults, and provides feedback to the mutation engine. The cluster's elements (MCUs) are mutually-independent. That is, they do not share any information of the fuzzing process. The fuzzing engine in the workstation, however, can leverage the feedback from the cluster in its own way, taking advantage of the workstation's plentiful computing resources.

Next, we describe how SHiFT supports the fuzz testing process, and our reasoning to solve specific challenges that the MCU platform poses for this novel approach.

3.5.1 Architecting firmware for fuzzing

SHiFT supports single (*S-C*) and dual-core (*D-C*) configurations to take advantage of modern MCUs' heterogeneous architectures. For each configuration, our design defines two mutually-isolated components in the MCU to provide high flexibility, and maintain a robust fuzzing platform.

The first component (Monitor in Figure 3) is the entry point of the semihosted fuzzer. This component manages the communication channel (input and feedback), MPU configuration, fault exceptions, and the initialization of the target. The second component (Target in Figure 3) executes the code under test, and collects coverage information. Monitor and Target are either tasks or cores. That is, they depend on SHiFT's configuration—whether it operates in *S-C* or *D-C*, respectively.

In *S-C* configuration, SHiFT uses RTOS primitives to create a monitor task, which in turn manages the target task. This target task can be deleted and recreated by the monitor task to exercise every testing input (standard mode), or can be reused for a configurable number of input tests, and later refreshed to improve performance (persistent mode) [14]. The *S-C* configuration is meant to test specific parts of the firmware, for example, user applications, libraries and drivers. This configuration is restricted in that the tested code must be compatible, or ported to run in a RTOS task (i.e., the target task).

In dual-core configuration, SHiFT maintains the same roles for the Monitor and the Target, but each one runs on a distinct core. This configuration is compatible exclusively with MCUs that implement two or more cores. This is a more flexible configuration in terms of testing because the Monitor and the Target are independent firmware running different libraries, RTOS, or even bare-metal designs. This configuration also supports the standard and persistent modes, but instead of deleting and recreating the target task, SHiFT reboots the target core to refresh it.

3.5.2 Delivering fuzzing input

SHiFT is highly-flexible in supporting various methods to deliver the fuzzing input to the target. The specific method is defined configuring small harness functions that extend the monitor and the target (i.e., configuration in §3.2). The simplest method, which resembles AFL running in a workstation, is by sharing the monitor's input buffer with the target (i.e.,

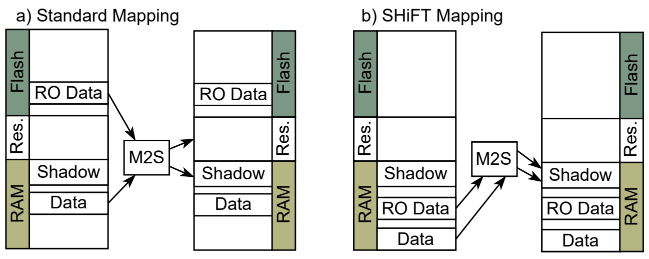

Figure 4: ASAN Mem-to-Shadow (M2S) computation in a MCU, a) shows an illegal mapping to a reserved memory area (Res.), and b) shows a valid mapping compatible with MCUs.

① in Figure 3). The most complex method, and key aspect of SHiFT's contributions, is by using real peripherals to send the fuzzing input from the monitor, and receive the fuzzing input in the target. This method supports and executes all the peripheral's MMIO, interrupts and DMA (i.e., ②, ③, and ④ in Figure 3) handled directly by the MCU hardware with high fidelity. For this method, the harness function in the monitor includes peripheral drivers to write the fuzzing input, and the target includes the original application's drivers and routines to read data from peripherals. Also, the tester needs to physically interconnect peripherals leveraging the development board (Figures 1 and 10). The reader is referred to our case study in §5.6.3 for a practical example.

3.5.3 Detecting and managing faults

SHiFT leverages the FreeRTOS-MPU kernel to detect out-of-bounds accesses and privilege mismatches. SHiFT also controls the Cortex-M traps for integer division-by-zero and unaligned accesses. For MCUs that integrate a floating-point unit (FPU), our design also handles exceptions derived from division-by-zero and invalid floating-point operations. SHiFT detects hangs by using RTOS primitives or watchdog timers, for *S-C* and *D-C* configurations, respectively.

Our design masters the Cortex-M exception model by defining specialized RTOS-independent handlers to systematically manage faults. This model differentiates exception conditions from usage faults (division-by-zero, unaligned accesses, and undefined instructions), hangs or timeouts, memory faults (MPU mismatches) and bus faults (e.g., illegal fetches) [48]. SHiFT uses the exception handlers to stop the target, and notify the monitor with the corresponding exit code using RTOS primitives. Notice that this exception model is a commodity on standard computers, but non-existing in embedded devices [59].

3.5.4 Supporting compiler instrumentation

SHiFT supports SANCOV, ASAN and optionally UBSAN run-time libraries for bare-metal and RTOS-based firmware. From a high-level perspective, the run-time library constitutes the implementation of callbacks that the compiler inserts into the code to trace the program counter on every basic block (BB), or to add memory checks on load and store operations to detect memory errors. To support the run-time libraries, our design tackles two fundamental constraints of the MCU: limited memory RAM, and the lack of MMU.

For SANCOV, our run-time design is inspired, but not limited, by AFL [1] with two particular optimizations in coping with the limited RAM, and returning the coverage feedback to the fuzzing workstation.

The first optimization reduces the size of the bitmap (i.e., RAM) that holds the number of hits of every edge[4]. We reduce it up to 1/8 of the original bitmap size (i.e., 64kB/8). This is feasible since MCU firmware is typically much smaller than regular software.

The second optimization adds an extra level of indirection into the bitmap to build the feedback dynamically. This is due to the observation that the bitmap hits are very sparse, making it highly inefficient to send the entire bitmap back for every fuzzing input. Thus, we only send back to the workstation the exercised edges and their hit counts instead of the entire bitmap.

For ASAN, the original algorithm assumes it is running on a Type I device, or a regular computer that supports virtual memory. Concretely, ASAN uses 1/8 of the virtual space for shadow memory, which is used to check whether a byte is safe to access [65]. This method is *incompatible* with MCUs, because these devices do not have an MMU to translate arbitrary addresses from the virtual space into the physical location of the shadow in RAM. Figure 4 explains this issue. This figure shows that the computation M2S for the Data segment is correct pointing to the shadow in RAM. However, for the Read-Only (RO) Data segment, the computation M2S points to a reserved area.

In our design, we solve this issue by relocating the RO Data segment into RAM (i.e., Figure 4.b), reserving 1/8 of the available RAM for shadow, and systematically choosing an `Offset` value for ASAN M2S. Without the assistance of an MMU, the modified instrumentation computes valid physical addresses for the shadow of Data and RO Data segments.

The addition of ASAN extends MPU protections to allow SHiFT to detect errors, including out-of-bounds on the heap and in global variables, use after free, illegal free, and double free. In the case of the stack, ASAN increases MPU protection granularity by adding checks for internal structures allocated in the stack region.

3.5.5 Selecting a communication channel

We analyzed multiple commodity interfaces of commercial MCUs. Table 1 summarizes the results of writing and reading operations for various interfaces. We also conducted a survey (Appendix §A) and determined that USART and USB are the most popular interfaces in commercial MCUS. Upon our

[4]We omit and adapt some AFL details to simplify our description.

Interface	Native Speed	Write [MB/s]	Read [MB/s]
Ethernet	100 Mbps	11.71	5.86
USB ST-Link	12 Mbps	0.57	0.49
USB-CDC	12 Mbps	1.06	0.97
UART	7.5 Mbaud	0.72	0.74

Table 1: Average speed for read and write operations on different interfaces of the NUCLEO-H743 development board

analysis, we selected the serial communication USB-CDC because it assures a trade-off between throughput and availability. Also, it is universally-accessible by the fuzzing workstation because drivers for serial ports are already available in every major OS.

4 Implementation

SHiFT's implementation includes, FreeRTOS-MPU extensions, run-times for SANCOV, ASAN and UBSAN, GCC compiler modifications, templates for *S-C* and *D-C* configurations including harness prototypes, a communication protocol and a customized serial proxy. The changes for GCC compiler are 2 LoC. The rest of our implementation constitutes of 2,140 LoC distributed on different SHiFT's components, which does not take into account the target firmware code. The proxy includes 600 lines of code (LoC) that we implemented exclusively on the fuzzing workstation.

4.1 FreeRTOS-MPU and kernel extensions

We selected FreeRTOS-MPU (V10.4.2) because it is arguably the most popular RTOS for Type II devices [20]. However, we used exclusively primitives available on any RTOS to make SHiFT easily portable. These primitives include methods to spawn and stop tasks, semaphores and notifications. Further, supporting the Cortex-M MPU is not mandatory, but highly desirable for better fault detection and robustness.

We extended the FreeRTOS-MPU kernel using the `application_defined_privileged_functions.h` file. This header file is used to define functions that the application writer wants to execute in privileged mode (i.e., syscalls). Thus, we defined wrappers for *malloc*, *free*, and *watchdog* handlers, which are used by the ASAN run-time and fuzzing templates. If the RTOS does not support the MPU, all the firmware is executed in the privileged mode and these wrappers are not necessary.

4.2 SANCOV and ASAN run-times

We used the GNU ARM Embedded Toolchain 10.3-2021.10 release [9], including the source code of the GCC compiler, libraries and run-times in the same package.

Our SANCOV run-time implements coverage measurements based on a modified version of AFL's edge tracking algorithm [36]. Our modifications account for the optimizations described in §3.5.4, which are mostly implemented in the `__sanitizer_cov_trace_pc` callback inserted by SANCOV to trace the program counter.

Configuration	Mem. area	Base Addr.	Size [kB]	Offset
Single-Core Monitor & Target	RAM	0x24000000	512	
	SHADOW	0x20000000	64	0x1B800000
	FLASH	0x08000000	1024	
Dual-Core Target	RAM	0x24000000	256	
	SHADOW	0x10000000	32	0x0B800000
	FLASH	0x08100000	512	
Dual-Core Monitor	RAM	0x24060000	128	
	FLASH	0x08000000	512	

Table 2: Memory layout to support ASAN instrumentation for the STM32H74x MCU family.

Our ASAN run-time is based on the `libsanitizer` source code. We make it compatible with RTOS-based and bare-metal designs by refactoring the run-time callbacks and removing the support for thread details and advanced memory allocators, which are OS-specific. The refactored callbacks include functions for (de-)initialization of the run-time and poisoned areas, reporting `load` and `store` violations, and memory (de-)allocators helpers. To support ASAN instrumentation, we re-compiled GCC with a custom `Offset` defined in `asan_mapping.h` and `arm.c` of GCC source code[5]. We also modified the firmware linker script, as described in Table 2, to accommodate the shadow, and relocate the data segments, according to the details in §3.5.4.

4.3 Communication protocol

We implemented a robust 2-message protocol to send the fuzzing input to the MCU, and to send back the feedback from the MCU. This protocol also includes a cyclic redundancy check (CRC) for error detection. On the MCU side, we built the protocol on top of a generic USB communication device class (CDC) example provided by the MCU vendor.

4.4 Serial proxy

The serial proxy abstracts the communication between the fuzzing engine and the MCU connected through a serial port (Figure 3). It acts as the target program maintaining compatibility with AFL, AFL++, and any other SoTA fuzzing engine that uses the same communication primitives (i.e., pipes and shared memory). We use libserialport [16], a cross-platform library, for communication between the proxy and the serial port in the fuzzing workstation.

5 Evaluation

We evaluated SHiFT to answer the following questions:

[5]GCC supports `Offset` as a command line option for KASAN, but not for ASAN. [22]

1. What is the compatibility of SHiFT with relevant embedded architectures? What are the requirements of SHiFT in terms of FLASH and RAM?

2. How does SHiFT's observability and fault detection capabilities compare to the SoTA?

3. How does SHiFT's fidelity compare to other solutions?, and How does reduced fidelity affect fuzz testing?

4. What is the fuzzing performance of SHiFT?, and how does SHiFT's unique characteristics support fuzz testing of complex hardware settings?

To answer 1), we empirically evaluated the compatibility of thirteen embedded architectures in §5.1, and analyzed the memory footprint of SHiFT in §5.2. To answer 2), we performed a fault detection analysis and compared to the SoTA in §5.3. To answer 3), we conducted a qualitative analysis of SHiFT fidelity comparing to the SoTA, and empirically analyzed the effects of low fidelity in fuzz testing in §5.4.1 and §5.4.2, respectively. Finally, to answer 4) we conducted a performance analysis in §5.5, and describe case studies in §5.6.

The default setting for our experiments includes a workstation equipped with an AMD Ryzen 3700x CPU, 32 GB of RAM, and Ubuntu 22.04. We used a wide range of development boards from ST-Microelectronics, Adafruit Industries, NXP, Microchip Semiconductors, Gigadevice, and Espressif Systems. Collectively, we evaluated 13 embedded architectures, including complete fuzzing campaigns on 6 different development boards from different vendors, and three architectures (i.e., ARMv7-M, ARMv8-M and Xtensa). Please refer to Table 3.

5.1 Compatibility analysis

For this analysis, we selected 13 popular embedded architectures according to market studies [4, 20]. Then, we verified their characteristics, toolchain and documentation considering SHiFT's design assumptions detailed in §3.1. Finally, we obtained the latest public toolchain, ported, compiled and tested SHiFT prototypes (eleven in total) considering the characteristics of each architecture, and the hardware that we have for validation.

The results in Table 3 demonstrate that our design has excellent compatibility since 100% (13) of the architectures support SANCOV, 92% (12) support software-based error detectors, and 77% (10) support hardware-based protections. The only exception is the 8bit AVR architecture which does not support software or hardware methods to detect faults. We estimate that porting SHiFT to another architecture requires on average 4 hours of effort for a single engineer, which includes the recompilation of GCC (about 90 minutes).

Architecture	**MPU**	**GCC**	**SANCOV**	**ASAN**	**UBSAN**	**Port MCU**
ARMv7-M	✓	9.3.1	✓	✓	✓	SMT32H745/H743 SAMD51, K66F
ARMv8-M	✓	9.3.1	✓	✓	✓	STM32L552
Xtensa	✓	8.4.0	✓	✓	✓	ESP32 WROM
MIPS M4K	MMU	8.3.1	✓		✓	PIC32MX795
MIPS MK64F	MMU	8.3.1	✓		✓	PIC32MZ2048
RISC-V	optional	9.2.0	✓		✓	GD32VF103CBT6
Renesas RX	✓	8.3	✓		✓	RSF562N8BDFP
Renesas RL	✓	11.1*	✓		✓	–
AVR		7.3.0	✓			Atmega2560
MSP430	optional	9.3.1	✓			–
ARC	optional	11.2.0	✓			–
Coldfire		9.3.0	✓			–
Power PC 400		9.3.0	✓	✓	✓	–

Table 3: Architecture compatibility with security hardware and instrumentation options. *only LLVM toolchain available for this architecture.

Configuration	**M. area**	**1/4 bmp**		**1/8 bmp**		**1/16 bmp**	
		kBytes	%	kBytes	%	kBytes	%
Single-Core	RAM	237.0	23.2	173.0	16.9	157.0	15.3
	FLASH	150.2	14.7	150.2	14.7	150.2	14.7
Dual-Core	RAM	189.2	18.5	157.2	15.4	141.2	13.8
	Flash	178.8	17.5	178.8	17.5	178.8	17.5

Table 4: Flash and RAM requirements for different fractions of the original (64k) AFL bitmap (bmp). % represents the usage of available Flash and RAM for the STM32H74x MCU.

5.2 Footprint analysis

In this section, we analyze the sizes of RAM and FLASH needed by SHiFT under single-core (*S-C*) and dual-core (*D-C*) configurations as well as different AFL bitmap size settings. The results in Table 4 demonstrate that SHiFT's footprint is moderate. It uses at most 23.2% and 17.5% of RAM and FLASH, respectively, out of the 1024kB provided by the MCU for each section. The moderate requirements make SHiFT compatible with actual embedded (i.e., constrained), or even more memory-limited scenarios. Also, note that the bitmap size is highly-adjustable to fit various characteristics of the fuzzing target.

5.3 Fault detection analysis

For this experiment, we configured SHiFT in S-C mode using the synthetic benchmark detailed in Appendix C as the target program. This benchmark contains artificial errors (detailed in Table 5) that the target executes upon a simple verification of conditional statements evaluated on a testing input. The goal of this test is to verify that SHiFT's hardware and software-based protections can detect, and report the firmware errors detailed in Table 5 according to our design. Simultaneously, we compared these results and notable SHiFT's testing features with re-hosting SoTA works according to their respective conceptual designs. For this particular analysis, we did not run our synthetic benchmark on each tool to avoid biased results due to wrong configurations, or assumptions not fulfilled by our benchmark.

Table 5 demonstrates that SHiFT provides comprehensive fault detection features, which collectively, cannot be supported by any SoTA tool. Note that the fault detection capabilities of each tool are not directly related to the vulnerabilities reported by them. This is because some reported vulnerabilities were identified and interpreted upon manually debugging general crashes that are observed later during firmware execution, but not when the actual fault happened.

5.4 Fidelity analysis

Fidelity is arguably the most important and difficult property to quantify in re-hosting and emulation-based scenarios [51, 52, 71]. There are works that propose tailored fidelity metrics to favor particular designs [76], or benchmarks meant to evaluate how well the soft-emulation technique mimics the real hardware (e.g., the unit test of P2IM [34]). However, all of those metrics or benchmarks are not applicable to SHiFT because we use the real hardware, thus comparing to soft-emulation would be unfair. Instead, we present a qualitative analysis inspired by Wright et al. [71], and we empirically explain the effects of low fidelity on dynamic firmware analysis.

5.4.1 Qualitative analysis

The qualitative analysis defines a 2-dimensional plane for the conceptual axes of memory and execution fidelity. The memory axis represents the state of any addressable resource of the MCU (i.e., RAM, FLASH, and peripherals). This axis increases granularity from black box to register. For black box, granularity is coarse, and the memory state is only verifiable externally. On the other hand, register means that the state of both, internal memory and registers, is similar to the real device. Execution fidelity axis denotes how close the emulated execution mirrors the behavior when firmware runs on the actual device. In black box, the external behavior is similar to the real device, but internally, there is no warranty of the functions or instructions that are executed. On the other end, cycle means that the emulated platform mimics the behavior of the real device at the CPU cycle level. For both axes, the perfect level is only achieved by the real device. Also, there is a blur between classification points since specific parts of the firmware can be executed at different levels of fidelity due to particular design considerations of each tool [71].

SHiFT achieves higher fidelity than emulation-based approaches, and slightly lower fidelity than GDBFuzz and uAFL. These tools, similar to SHiFT, use real hardware, and do not require instrumentation that can affect execution fidelity (Figure 5). We also consider that a perfect solution does not exist yet because, for example, GDBFuzz halts the processor continuously, and uAFL adds harnesses in the firmware to hold and provide fuzzing input. Thus, both tools cannot achieve cycle execution accuracy. [6]

[6]We updated Wright's work [71], with the SoTA and corrected a miss. The author confused the fuzzing phase of P2IM with its model instantiation.

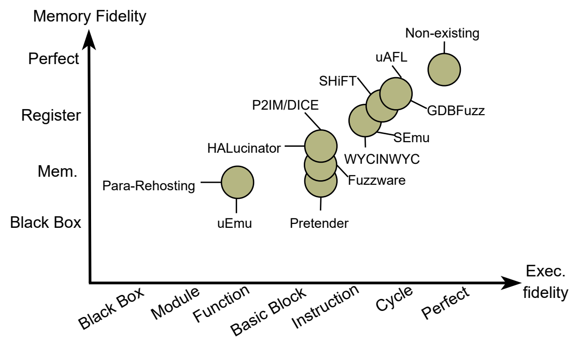

Figure 5: SHiFT's fidelity compared with the SoTA.

5.4.2 Impact of fidelity on dynamic analysis

For this analysis, we selected P2IM and Fuzzware, two prominent re-hosting solutions, to explain the fidelity effects on dynamic analysis when compared to hardware-based solutions such as SHiFT. We used the synthetic benchmark as a control test, and we specifically concentrate on the coverage tracking and false path exploration effects.

We observed that fuzzing engines struggle in tracking code coverage when the firmware is forced to consume input with sophisticated, but low-fidelity, methods. For example, for the synthetic benchmark, SHiFT identified all the bugs in less than 6 minutes, whereas P2IM identified only three in 24 hours (Table 8). We verified this effect by coverage, and not by fault detection capability. A similar effect was observed by the authors of Fuzzware. They found their sophisticated "bit-granular input overhead reduction" ineffective because it "conflicts with the byte-level heuristics that drive the fuzzer's input mutation process" [63].

We also observed that aggressive strategies to increase code coverage can violate hardware invariants, increase false positives, and put additional burden on the tester. For example, Fuzzware is optimized for code coverage by providing "optimized" fuzzing input when firmware reads from peripheral's registers that are determined to affect firmware state. On the other hand, P2IM maps primitive models that mimics the peripheral's register behavior, and exclusively provide fuzzing input for Data Register (DR) read operations.

We observed that the conservative P2IM approach–in this particular case–honors the expected hardware invariant by returning a previously written value on a Configuration Register (CR), which is evaluated in a critical conditional statement. Fuzzware, however, violates this hardware invariant by assigning a "Bitextract" model (i.e., a masked fuzzing input) to mutate specific bits of the CR and increase coverage factitiously. This violation makes Fuzzware perform poorly by majorly exploring false paths, reporting false positives and not reaching the target function of the synthetic benchmark. Nevertheless, we need to mention that for different firmware images and conditions, P2IM heuristic approach can also mis-

Error detection/Feature	SHiFT HW	SHiFT SW	SHiFT Combined	GDBFuzz	Wycinwyc	P2IM/ DICE	uAFL	HALucinator	Fuzzware	Para-rehosting	Pretender
Time Out	✓	✓	✓	✓		✓	✓				✓
NULL dereference	✓	✓	✓		✓	✓	✓	✓		✓	
Div By Zero	✓	✓	✓		✓	✓	✓	✓		✓	
Unaligned Access	✓		✓								
Use After Free		✓	✓		✓			✓		✓	
Double Free		✓	✓		✓			✓		✓	
Seg. Fault	✓		✓		✓	✓	✓	✓	✓		
Stack Overflow	✓	✓	✓		✓				✓	✓	✓
Glb Var Overflow	✓	✓	✓							✓	
Heap Overflow		✓	✓		✓			✓		✓	
Undefined instruction	✓		✓		✓	✓	✓	✓	✓	✓	
DMA	✓		✓	✓		✓✗	✓	✓✗			
Complex Peripherals	✓		✓	✓			✓		✓✗	✓✗	✓✗
Real-time Operations	✓		✓	✓✗			✓				
Interrupts	✓		✓	✓	✓✗	✓✗	✓	✓✗	✓✗	✓✗	✓✗

✓: full support ✓✗: partial support

Table 5: Details and comparison of error detection and testing features according to hardware and software-based characteristics of SHiFT's design.

Fuzzing mode	Native AFL++	SHiFT S-C	SHiFT D-C
Standard	4.9	4.8	0.41
Persistent	23.5	5.9	5.1
Standard With ASAN	*1.9*	*4.6*	*0.32*
Persistent With ASAN	22.7	5.7	4.1

Table 6: End-to-end fuzzing performance in [kRun/s] of (1) vanilla AFL++ running in the workstation, a single instance of SHiFT for (2) single-core (S-C) and (3) dual-core (D-C) configurations.

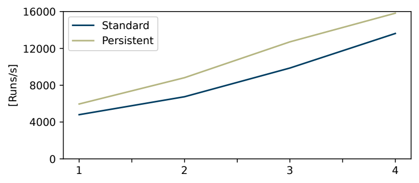

Figure 6: SHiFT's cluster test for *S-C* configuration.

classify peripheral registers, violate hardware invariants, and reflect similar issues as Fuzzware. These limitations has been already acknowledged by P2IM's authors [34].

5.5 Performance analysis

For this analysis, we used the same synthetic benchmark running on SHiFT as an end-to-end fuzzing framework to measure its raw performance in terms of runs/sec on different configurations, including the cluster setup.

Despite the huge computing power advantage of the workstation, the results show that SHiFT performance is comparable and, in some cases, superior to our workstation running AFL++ natively (i.e., without emulation). Specifically, SHiFT is 2.42 times faster than the workstation in standard mode (i.e., creating a new target for every fuzzing input) when running with ASAN instrumentation, as detailed in Table 6. This impressive result is obtained mainly thanks to the FreeRTOS spawning process, and our MCU-specific ASAN run-time. These two components are leaner and lighter than their workstation counterparts without sacrificing equivalent functionality for real embedded applications.

In the case of the *D-C* operation, the results are merely informative since there is no equivalent operation mode of AFL++ in a workstation. Nevertheless, note that in this mode, SHiFT's performance is still significant since it can fuzz and reboot a complete core more than 410 times per second. Also, note that performance decreases significantly in this mode compared with the *S-C* mode because the target runs in the slower CM4 core of our MCU.

For the cluster performance, we analyzed the *S-C* in the standard and persistent mode for the settings of one to four development boards connected simultaneously. SHiFT boosts performance linearly with two or more MCUs, whereas a single board performs relatively better when working alone, as depicted in Figure 6. We attribute this result to the nature of the USB protocol whose low-level analysis we considered out-of-the-scope of this work.

5.6 Case studies

This section presents case studies of SHiFT performing fuzz testing campaigns on firmware used in industrial and commercial devices. The tested firmware includes samples previously tested by other works, and new firmware collected from open-source projects. Each firmware was tested 10 times for 24 hours using a fixed seed input[7], and a single development board for the hardware-based solutions, i.e., SHiFT and GDBFuzz. We used the AFL++ mutation engine for all tools, except for P2IM because it is tightly-coupled with a customized AFL. Porting P2IM to AFL++ is a non-trivial task, thus we consider it out of the scope of this work. Thanks to SHiFT's unique characteristics, we identified five previously unknown vulnerabilities as shown in Table 7.

[7]We used the UTF-8 encoded string "testinput".

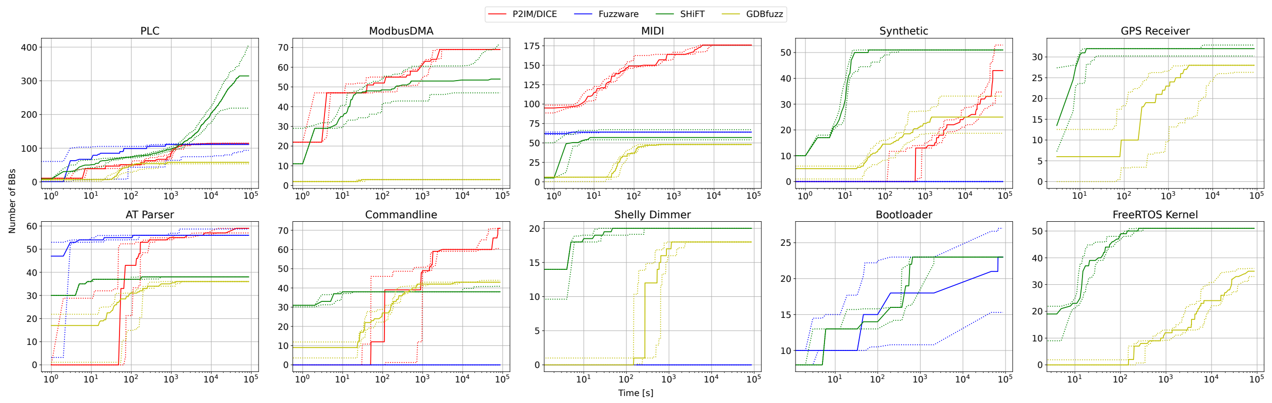

Figure 7: Basic block (BB) coverage over 24-hour campaigns for SHiFT compared to P2IM/DICE, Fuzzware and GDBFuzz. Shown are the median and 95% confidence intervals.

Firmware	Vulnerability	CWE	Instances
Shelly Dimmer	Divide by zero	369	3
Bootloader	Buffer overflow	120	1
FreeRTOS K.	Improper handling of insufficient privileges	274	1

Table 7: Vulnerabilities identified by SHiFT. Each instance is categorized according to the Common Weakness Enumeration (CWE).

5.6.1 Fuzzing libraries and full-stack firmware

SHiFT is highly flexible in that it allows exercising specific functions of the firmware (i.e., cases #1, and #4 to #7 in Table 8) by delivering the fuzzing input through a shared memory area. SHiFT can also exercise the complete software stack including, drivers, ISR, and the user application, which uses DMA operations to get data from peripherals (i.e., cases #2 and #3).

For the aforementioned test cases, SHiFT outperforms the SoTA[8] by up to two orders of magnitude in speed, while detecting all the previously reported vulnerabilities (i.e., nine in total on firmware #1, #2 and #3) with no false positives.

In terms of basic blocks coverage, SHiFT outperforms GDBFuzz in all cases, except underperforming slightly for the CommandLine firmware. In comparison, the performance of soft-emulation solutions widely fluctuate between outperforming (e.g., case #7 for P2IM) and totally falling short (e.g., cases #4 and 7 for Fuzzware), when compared with SHiFT. We consider this result as expected because of different design choices of soft-emulation approaches that affect fidelity and the fuzzing performance. The reader is also referred to §5.4 for more details about this phenomenon.

Note that all our observations demonstrated statistical significance through the Mann-Whitney U test ($p<0.05$). Also, the reported basic blocks were filtered according to the target functions instrumented by SHiFT. This procedure can slightly benefit SHiFT and GDBFuzz, but allows comparability by "normalizing" the code sections under analysis. This is necessary due to the slightly different targets under analysis for hardware-based and soft-emulation approaches (i.e., soft-emulation usually consider the whole binary, while GDBFuzz relies on a user-specified entry function, and SHiFT is based on source-code instrumentation).

[8]We exclude from the SoTA High-level emulation (HLE) approaches [26, 64] because they require considerable engineering effort and do not execute the whole firmware.

5.6.2 Finding insecure conditions and post-silicon bugs

We fuzz tested the full-stack of the dimmer depicted in Figure 8. This firmware (#8 in Table 8) uses communication peripherals, GPIOs, real-time events and interrupts. Under this configuration, SHiFT identified three previously unknown vulnerabilities that can be *realistically observed in a deployed device*. The three vulnerabilities are similar in nature to the code in Listing 1. Specifically, function `get_active_power`, which is accessible remotely, is vulnerable to a division-by-zero in line #7, under these specific conditions: 1) the dimmer receives a command to report the active power, 2) condition in line #3 is `false` because of ADC operation, and 3) `voltage_max_period` is zero.

```
1   static uint32_t get_active_power(void)
2   {
3       if ((adc_data[0] == 0) || (adc_data[1] == 0))
4           return 0;
5       else
6           return (880373 * 15.5 * 556) /
7               voltage_max_period *
8               current_total_mag_period;
9   }
```

Listing 1: Vulnerable code fragment of the dimmer firmware

Conditions 1) and 2) are easily satisfied by SHiFT features because it supports firmware reading the ADC through DMA, receiving a remote command through the USART, and triggering the corresponding interrupt service routine. Condition 3) occurs only when the dimmer receives a command before the zero crossing routine is executed at least once. We verified this condition is met only on the actual device on two scenarios: a) when a command is received on the first 8.3 [ms] of operation of the dimmer (i.e., a time sensitive operation), or b) when the zero-crossing module is disconnected or fails

Ref	**#**	**Firmware**	**Method**	**Board**	**SHiFT**			**P2IM/DICE**				**Fuzware**				**GDBFuzz**			
					[r/s]	**TP**	**FP**	**[r/s]**	**SU**	**TP**	**FP**	**[r/s]**	**SU**	**TP**	**FP**	**[r/s]**	**SU**	**TP**	**FP**
P2IM	1	PLC	Function call	H743	3100	4	0	32.7	×95	4	4	30.9	×100	4	2	70	×44	4	0
DICE	2	Modbus	Full-stack DMA	H743	1800	3	0	41.6	×43	3	2	NB	-	-	-	327	×6	0	0
	3	Midi	Full-stack DMA	H743	1200	2	0	59.9	×20	2	0	208	×6	0	0	37	×32	0	0
SHiFT	4	Synthetic	Function call	H743	4800	11	0	94.5	×51	3	1	85.9	×55	0	10	32.1	×150	2	0
	5	GPS Receiver	Function call	ESP32	380	0	0	NB	-	-	-	NB	-	-	-	170	×2	0	0
	6	AT parser	Function call	SAMD51	276	0	0	44.1	×6	0	1	53.5	×5	0	1	55	×5	0	1
	7	Command line	Function call	K66F	233	0	0	63.5	×4	0	1	321.9	×1	0	1	245	×1	0	1
	8	Shelly Dimmer	Real-time DMA	H743	1148	3	0	NB	-	-	-	321.3	×4	0	1	24.5	×25	0	4
	9	Bootloader	Baremetal	H745	170	1	0	NB	-	-	-	89	×2	0	0	NB	-	-	-
	10	FreeRTOS K.	Function call	L552	3750	1	0	NB	-	-	-	NB	-	-	-	43	×86	0	0

Table 8: 24-hour fuzzing campaigns of SHiFT on real firmware and a *Synthetic* benchmark compared to the SoTA. **SU:** SHiFT SpeedUp (average), **TP:** True Positives (median), **FP:** False Positives (median), **NB:** No Bootstrap. New **TPs** observed on firmware # 8, 9, 10.

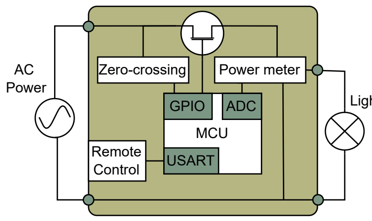

Figure 8: Simplified architecture of an IoT dimmer fuzz tested with SHiFT.

(i.e., a peripheral interface issue). Thanks to SHiFT unique characteristics a tester can fuzz test firmware combined with specific hardware scenarios (e.g., a disconnected cable), and reason about unsafe/insecure conditions that can happen in a deployed device, but cannot be observed in an emulator.

Also, further analysis demonstrated that SHiFT helped to identify a documented post-silicon bug that we were not aware of beforehand. In particular, we observed that the division by zero is only caught by SHiFT's UBSAN instrumentation, but not by the real FPU when configured to trigger exceptions on this type of errors. Thus, GDBFuzz and SHiFT, this last configured without UBSAN, were not able to identify the division by zero. The reason for this unexpected behavior is that the FPU's interrupt line is not present on the NVIC silicon of the MCUs STM32h743 and 745 [67]. Notice, that post-silicon issues like this cannot be identified by emulated platforms either, because this issue only affects the real FPU of a particular MCU family.

5.6.3 Fuzzing bare-metal firmware

The bootloader firmware (#9 in Table 8) demonstrates the flexibility of SHiFT for testing firmware that implement baremetal designs and uses complex peripherals. The bootloader implements a full-state protocol that remotely manages the flashing and execution of the main MCU's application through the CAN bus.

We used the *D-C* configuration of SHiFT to test the bootloader. We interconnected the monitor (Cortex-M7) and the target (Cortex-M4) using two on-chip CAN bus peripherals and external CAN bus transceivers as detailed in Appendix B. The fuzzing setup operates strictly under the CAN bus specifications (i.e., frame format, payload size, and speed), and supports all the asynchronous events derived from this bus operation. We commented out the bootloader's operations that write into the MCU's flash to avoid wearing out this memory during the fuzzing campaign.

Thanks to SHiFT unique setup, we discovered a vulnerability with security implications in the communication protocol of the bootloader. Specifically, this protocol temporarily accumulates in RAM chunks of data up to the size of a flash page before writing it into the MCU's flash. Unfortunately, this routine uses the data length code (DLC) embedded into the CAN bus frame to weakly verify the length of the received data chunks (i.e., the routine assumes the DLC is always a submultiple of the flash page's size). If the DLC is not a submultiple of the flash page's size, the bootloader routine keeps storing the incoming chunks of data overflowing the temporal buffer, and potentially compromising critical memory sections of the firmware.

5.6.4 Finding a zero-day in the FreeRTOS kernel

This bug was identified when we ported SHiFT to the NUCLEO-L552 board, which implements a Cortex-M33 MCU (#10 in Table 8). We used our synthetic benchmark as a control test on this board and observed that the FreeRTOS `TaskYield` syscall triggered a systematic MPU violation. This syscall is used by SHiFT's monitor to switch context when fuzzing a target. Upon analysis, we identified that the FreeRTOS port for the entire ARMv8-M architecture, including the Cortex-M33, fails to obtain privileged execution level before writing the ARM `Interrupt Control and State Register` [3]. This register is only writable on privileged mode, and is used to trigger a `PendSV` interrupt that handles a request to switch context. We consider that this vulnerability has security implications because when triggered causes a denial-of-service. We responsibly reported this issue to the FreeRTOS developers, who acknowledged and patched it.

This zero-day highlights the benefits of automated testing on real hardware even when (FreeRTOS-)developers have access to source code and traditional debugging tools. Notice that soft-emulated platforms can potentially help to detect this bug as well. However, until the time of this writing, P2IM, Fuzzware and other tools based on QEMU or the Unicorn engine do not fully support the Cortex-M33 or the Xtensa architectures, just to mention a few. This issue can be considered only an engineering limitation, but in reality, it exposes a fundamental challenge (i.e., hardware diversity) of embedded platforms that is not easily addressed by soft-emulation.

6 Discussion

Can SHiFT replace soft-emulation? The answer to this question is no. Soft-emulation is well-known for its independence from the physical platform, which in turn facilitates advanced analysis with much flexibility and scalability. For instance, SHiFT provides high fidelity in exchange for a relatively less scalable cluster-based design. Soft-emulation, on the other hand, can leverage multiple instances (e.g., taking advantage of cloud computing) to reduce the time needed to explore the input space. However, it cannot improve fidelity. Consequently, we argue that soft-emulation and hardware-based solutions like SHiFT are rather complementary than competing approaches.

Testing on hardware caveats: Fuzz testing on embedded development boards needs extra precaution. This process stresses the hardware, can damage the system, or force it into unsafe states. Therefore, the tester has to implement mitigations to, for example, prevent wearing-out the MCU's FLASH by repetitive writing operations or avoid testing devices connected to physical actuators that can behave unsafely.

Binary-only scenarios: SHiFT's design assumes access to the firmware's source code. However, testing firmware in a binary-only form is always valuable when the embedded application cannot tolerate instrumentation, when developers link first-party source code with third-party blobs, or simply when source code is not available. In this scenario, SHiFT can borrow design ideas from previous works (e.g., [31], and [47]) to acquire coverage information from blobs using debugging or tracing components provided by the Cortex-M architecture (e.g., the ARM-embedded trace macrocell (ETM), or the embedded trace buffer (ETB) [19]). We consider supporting these hardware facilities out of the scope for this work.

Semihosting beyond fuzz testing: SHiFT provides strong evidence and a reference design that future work can leverage to perform dynamic analysis in MCUs. We envision applications where semihosting catalyzes other verification techniques that benefit from high fidelity (e.g., concolic execution). In this scenario, executing the firmware in an actual device provides accurate concrete values to mitigate state explosion and false positives (i.e., two common pitfalls of concolic techniques). Meanwhile, workstations can perform computing-intensive, but generic, operations (e.g., constraint solvers) on behalf of the MCU.

7 Related Work

Soft-emulation and HLE: These approaches eliminate the hardware and physical layers through using emulated models or High-level Emulation (HLE). For example, Firmadyne [25] uses QEMU to emulate a generic kernel with the filesystem of the firmware to perform user space analysis. FirmAE [73], FIRM-AFL [74] and Costin's work [29] adopted a similar approach. These works are only applicable to firmware using a specific kernel or filesystem since they rely on the Linux kernel abstractions. To analyze monolithic firmware, other works leverage emulated models of peripheral behavior at different abstraction layers [26, 29, 34, 54].

HALucinator [26] hooks calls into vendor-specific Hardware Abstraction Layer functions to implement the HLE paradigm within a generic emulator. Safirefuzz [64] proposes a similar approach but removes emulation by near-native execution of binaries as Linux user space processes in an ARM exclusive design. Para-rehosting [46] also executes firmware natively in Linux using HLE principles and source code. This last approach supports instrumentation-based sanitizers, but contrary to SHiFT that executes the entire firmware in the MCU, Para-rehosting executes sections of the firmware in a workstation with reduced fidelity.

P2IM [34] and μEMU [75] model MMIO access patterns. DICE [54] extends emulators to support basic DMA operations, and SEmu [76] uses NLP to process documentation and improve models. Unlike SHiFT, soft-emulation analysis suffers from false positives during the emulation. It is also limited in real-time analysis, precise interrupt handling, and advanced DMA support. Except DICE [54], and partially HALucinator [26], DMA support is considered orthogonal, or out of scope by most of the re-hosting works.

HiL: These approaches avoid the need for hardware abstractions by forwarding hardware accesses to a physical device during the emulation. In particular, AVATAR [72] orchestrates the execution of an emulator and implements an expensive synchronization to forward the I/O accesses from the emulator to the real device. SURROGATES [44] improves the forwarding performance through an FPGA-based interface and enables near-real-time analysis. PROSPECT [43] adopts a similar design with AVATAR and SURROGATES, but extends to Linux-based systems. It forwards system calls that are likely to access peripherals to the physical device to achieve a partial emulation during the fuzzing process.

Compared with SHiFT, HiL approaches are limited in flexibility. Specific debugging interface, or tailored hardware might be needed. It is also difficult to synchronize asynchronous or real-time events between the device and the emulation environment because these solutions halt the processor while synchronizing.

To improve performance and reduce the I/O forwarding latency, later works adopt similar design with different optimizations [32, 37, 58, 61, 70]. Charm [70] uses USB 3.0

to forward MMIO to a physical device to analyze Android device drivers for ARM in a virtualized environment. PRETENDER [32] records hardware behavior to automate the modeling of MMIO and interrupt-driven hardware peripherals. Kammerstetter [42] caches peripheral device communication and provides run-time program state approximation. Frankenstein [61] captures live snapshots of the physical hardware states during normal operation and integrates them to the emulated environment to analyze Bluetooth firmware. FIRMCORN [37] optimizes the virtual execution environment by hooking hardware-specific functions to avoid program halts and crashes.

These works sacrifice a complete real-time analysis, and can only support a specific type of hardware and software. Alternatively, SHiFT solves these problems executing the firmware entirely in an MCU, removing complex synchronization and dependencies on tailored hardware.

Similar to SHiFT, GDBFuzz and uAFL [31,47] are fuzzing frameworks that execute the code on the MCU. However, for collecting coverage SHiFT uses SANCOV instrumentation, whereas GDBFuzz uses breakpoints, and uAFL uses the ARM ETM. Their dependency on specific debugging facilities limits their applicability (e.g., uAFL only works for ARM). Also, GDBFuzz and uAFL do not support software sanitizers as SHiFT does, and they implement the fuzzing “manager” (i.e., SHiFT’s Monitor in Figure 3) in the workstation. Thus, uAFL and GDBFuzz are classic HiL with reduced fault detection capabilities and slower performance.

Symbolic abstraction: These approaches emulate software layers and consider all the values from hardware to be symbolic. For example, FIE [30] uses KLEE to symbolically execute the firmware. It assumes every peripheral is capable of returning the full range of possible values, which over-approximates the hardware capabilities. This leads to false positive and can cause state explosion even for a small firmware.

To increase scalability, later works explored different techniques [39, 66]. Firmalice [66] leverages program slicing and aims at automatic analysis of binary firmware. FirmUSB [39] develops targeting algorithms and uses domain knowledge to speed up the unconstrained symbolic execution for USB firmware. Fuzzware [63] uses symbolic execution to determine the meaningful parts of the hardware-generated value. Lealaps [23] calculates the expected values for peripheral registers using symbolic execution, and steers the concrete execution on the fly. Jetset [41] uses guided symbolic execution to create peripheral models that make firmware to boot and execute till a particular point of interest.

Symbolic abstraction based analysis may also involve hardware-in-the-loop. Inception [28] performs symbolic execution to handle different levels of memory abstractions, interaction with peripherals, and interrupts. It then uses a JTAG debugger to redirect memory accesses to the real hardware. While this design reduces differences with real execution, collecting the interrupts from the real hardware suffers from inconsistencies. Compared to SHiFT, the above analysis is computation intensive and limited to small size firmware.

8 Conclusion

In this paper, we presented SHiFT, a semi-hosted fuzz testing framework to enable dynamic analysis of MCU firmware with high fidelity and support for complex peripherals, interrupts, real-time operations, and DMA. SHiFT can detect faults that previous rehosting approaches have overlooked or lack support. We demonstrated empirically that boosting coverage without maintaining fidelity is not enough, or even worse for firmware fuzzing. Our evaluation shows that SHiFT is compatible with multiple embedded architectures, and requires only moderate memory resources. SHiFT achieved a high-speed performance comparable with a workstation running AFL++ natively, and superior to that in some cases. When testing real firmware, our framework exposed five previously-unknown vulnerabilities. We deeply discussed SHiFT’s novel characteristics, and fairly compared to soft-emulation approaches. Finally, this work lifted the constraints of running security analysis natively in MCUs, and provided a reference design for future dynamic analysis in resource-constrained devices.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd for their insightful comments. This project was partially-supported by the National Science Foundation (Grant#: 2031390, “Rethinking Fuzzing for Security”). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Availability

Prototypes and source code available at https://github.com/RiS3-Lab/SHiFT

References

[1] American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[2] ARM Compiler toolchain Developing Software for ARM Processors. https://developer.arm.com/documentation/dui0471/i/semihosting/what-is-semihosting-?lang=en.

[3] Arm Cortex-M33 Devices Generic User Guide r0p4. https://developer.arm.com/documentation/100235/0004/the-cortex-m33-peripherals/syste

m-control-block/interrupt-control-and-state-register?lang=en.

[4] Arm's market share targets 2028. https://www.statista.com/statistics/1132112/arm-market-share-targets/.

[5] Bernd Schmidt - Re: Add fuzzing coverage support. https://gcc.gnu.org/legacy-ml/gcc-patches/2015-12/msg00307.html.

[6] Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[7] FreeRTOS-MPU - ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit support in FreeRTOS. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[8] GCC 4.8 Release Series — Changes, New Features, and Fixes - GNU Project. https://gcc.gnu.org/gcc-4.8/changes.html.

[9] GNU Arm Embedded Toolchain 10.3-2021.10 : GNU Arm Embedded Toolchain. https://launchpad.net/gcc-arm-embedded/+announcement/30539/+index.

[10] Product Selection Tools | Microchip Technology. https://www.microchip.com/en-us/products/selection-tools.

[11] Product Selector | NXP Semiconductors. https://www.nxp.com/products/product-selector:PRODUCT-SELECTOR.

[12] Semihosting - SEGGER Wiki. https://wiki.segger.com/Semihosting.

[13] ST-MCU-FINDER-PC - STM32 and STM8 product finder for desktops - STMicroelectronics. https://www.st.com/en/development-tools/st-mcu-finder-pc.html.

[14] New in AFL: Persistent mode. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html, June 2015.

[15] What is Semihosting? https://community.nxp.com/t5/LPCXpresso-IDE-FAQs/What-is-Semihosting/m-p/475390#M143, April 2016.

[16] Sigrokproject/libserialport. sigrok, December 2021.

[17] Fuzzware Experiments. fuzzware-fuzzer, March 2023.

[18] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46, 2019.

[19] ARM Ltd. Understanding Trace. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Understanding%20Trace.pdf, March 2020.

[20] Aspencore. 2019 Embedded Markets Study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, March 2019.

[21] Katharina Bogad and Manuel Huber. Harzer Roller: Linker-Based Instrumentation for Enhanced Embedded Security Testing. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, ROOTS'19, pages 1–9, New York, NY, USA, November 2019. Association for Computing Machinery.

[22] Julian Brown. KASAN should work even back-end not porting anything. https://gcc.gnu.org/pipermail/gcc-cvs/2020-August/317709.html, Mon Aug 3 22:56:06 GMT 2020.

[23] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 746–759, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, page 2021. San Francisco, 2011.

[25] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[26] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. {HALucinator}: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218, 2020.

[27] Forrester Consulting. State Of Enterprise IoT Security In North America: Unmanaged And Unsecured. Technical report, 2019.

[28] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *27th USENIX*

Security Symposium (USENIX Security 18), pages 309–326, 2018.

[29] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 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*, pages 437–448, 2016.

[30] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium*, pages 463–478, 2013.

[31] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing Embedded Systems using Debug Interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1031–1042, Seattle WA USA, July 2023. ACM.

[32] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 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, 2019. USENIX Association.

[33] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurelien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling security analyses of embedded systems via rehosting. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2021.

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

[35] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development (SecDev)*, pages 23–30, September 2020.

[36] Google. AFL Technical details. https://github.com/google/AFL/blob/master/docs/technical_details.txt, July 2019.

[37] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution. *IEEE Access*, 8:29826–29841, 2020.

[38] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application memory isolation on ultra-low-power MCUs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 127–132, Boston, MA, July 2018. USENIX Association.

[39] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2245–2262, 2017.

[40] Ponemon Institute. The State of IoT/OT Cybersecurity in the Enterprise. Technical report, 2021.

[41] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 321–338, 2021.

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

[43] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pages 329–340, Kyoto Japan, June 2014. ACM.

[44] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, page 7, Washington, D.C., August 2015. USENIX Association.

[45] Abraham Peedikayil Kuruvila, Ioannis Zografopoulos, Kanad Basu, and Charalambos Konstantinou. Hardware-assisted detection of firmware attacks in inverter-based cyberphysical microgrids. *International Journal of Electrical Power & Energy Systems*, 132:107150, 2021.

[46] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[47] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. uAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1–12, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Arm Ltd. ARM v7-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0403/latest/.

[49] Federico Maggi, Marcello Pogliani, and P Milano. Attacks on Smart Manufacturing Systems. *Trend Micro Research: Shibuya, Japan*, pages 1–60, 2020.

[50] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, pages 122–132, New York, NY, USA, July 2020. Association for Computing Machinery.

[51] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 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 XVII, pages 337–348, New York, NY, USA, 2012. Association for Computing Machinery.

[52] Lorenzo Martignoni, Roberto Paleari, Alessandro Reina, Giampaolo Fresi Roglia, and Danilo Bruschi. A methodology for testing CPU emulators. *ACM Trans. Softw. Eng. Methodol.*, 22(4), October 2013.

[53] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. D-Box: DMA-enabled Compartmentalization for Embedded Applications. In *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[54] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1938–1954. IEEE, 2021.

[55] ST Microelectronics. STM32H7 Nucleo-144 boards user manual. https://www.st.com/resource/en/user_manual/um2408-stm32h7-nucleo144-boards-mb1363-stmicroelectronics.pdf.

[56] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91), 2015.

[57] Subhasish Mitra, Sanjit A. Seshia, and Nicola Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Design Automation Conference*, pages 12–17, June 2010.

[58] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar ^{2} : A Multi-Target Orchestration Platform. In *Proceedings 2018 Workshop on Binary Analysis Research*, San Diego, CA, 2018. Internet Society.

[59] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[60] Qyriad. Fusée Gelée. https://github.com/Qyriad/fusee-launcher/blob/3b1b2bcca1b0e1f295e376b1dd8a1e582b18f41d/report/fusee_gelee.md, December 2021.

[61] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[62] Majid Salehi, Danny Hughes, and Bruno Crispo. μSBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 381–395, 2020.

[63] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[64] Lukas Seidel, Dominik Maier, and Marius Muench. Forming faster firmware fuzzers. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, pages 2903–2920, USA, August 2023. USENIX Association.

[65] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, June 2012. USENIX Association.

[66] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[67] ST Microelectronics. STM32H7x errata sheet. `https://www.st.com/resource/en/errata_sheet/es0392-stm32h742x743xig-st32h750xb-and-stm32h753xi-device-errata-stmicroelectronics.pdf`, 2023.

[68] Ruimin Sun, Alejandro Mera, Long Lu, and David Choffnes. SoK: Attacks on industrial control logic and formal verification-based defenses. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 385–402, Los Alamitos, CA, USA, September 2021. IEEE Computer Society.

[69] Zhichuang Sun, Bo Feng, L. Lu, and S. Jha. OAT: Attesting operation integrity of embedded devices. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449, 2020.

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

[71] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys*, 54(1), January 2021.

[72] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[73] Hangwei Zhang, Kai Lu, Xu Zhou, Yin Qidi, Pengfei Wang, and Tai Yue. SIoTFuzzer: Fuzzing Web Interface in IoT Firmware via Stateful Message Generation. *Applied Sciences*, 11:3120, April 2021.

[74] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[75] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2007–2024, 2021.

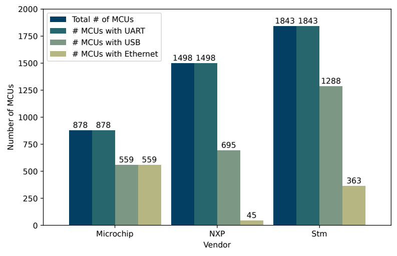

Figure 9: Communication interfaces availability on the MCU portfolio of top MCU vendors [10, 11, 13] May 2022.

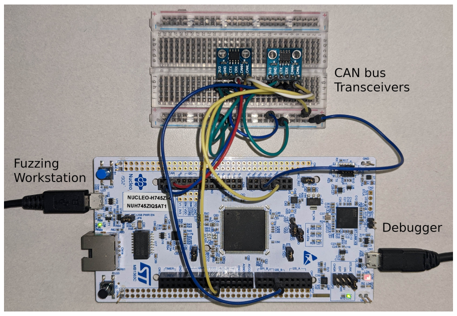

Figure 10: SHiFT setup for fuzz testing firmware with CAN bus.

[76] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 3269–3283, New York, NY, USA, November 2022. Association for Computing Machinery.

Appendix A Interface availability

Appendix B SHiFT fuzzing CAN bus

Appendix C Testing program

The following listing includes some examples of the synthetic bugs of our benchmark.

```c
uint32_t bufferGlobal[10];
int test(uint8_t *buf, uint32_t size)
{
    uint8_t *localbuff;
    uint32_t arr32[4];
    uint16_t *ptr16;
    if(size<7) return FAULT_NONE_RTOS; //normal execution
    if(buf[0]=='H' && buf[1]=='A' && buf[2]=='N' && buf[3]=='G') {
        buf[100]='T';
        while(1);  //Hang}
    else if(buf[0]=='S' && buf[1]=='E' && buf[2]=='G'){
        bufferGlobal[0]++; //Segmentation fault}
    else if(buf[0]=='N' && buf[1]=='U' && buf[2]=='L' && buf[2]=='L'){
        localbuff = 0x00;
        buf[0] = *localbuff; //Null dereference }
    ..
    return FAULT_NONE_RTOS;
}
```