DRIFT: Debug-based Trace Inference for Firmware Testing

Changming Liu
Northeastern University
Boston, MA, USA

Alejandro Mera
Northeastern University
Boston, MA, USA

Meng Xu
University of Waterloo
Waterloo, ON, Canada

Engin Kirda
Northeastern University
Boston, MA, USA

***Abstract*—Binary firmware fuzzing has garnered attention in recent years. Compared to source-code-based approaches, binary approaches require less semantic information and are therefore more applicable. This is particularly relevant in firmware analysis, as most firmware vendors distribute only binaries, withholding source code due to proprietary concerns.**

Pivoting away from the traditional hardware-in-the-loop (HiL) methodology, researchers are exploring more efficient ways to engage real hardware for fuzzing. However, existing approaches have inherent drawbacks, such as reliance on high-end hardware features, inability to recover complete coverage, and slow execution speeds. We propose DRIFT, a novel approach for on-device binary firmware testing that follows the semihosting methodology. DRIFT addresses all the aforementioned drawbacks. The core insight of DRIFT is to use the Debug Monitor (DM) for firmware fuzzing. DM is a Arm Cortex-M CPU feature that allows triggering interrupt when a breakpoint is hit. Through chaining the DM interrupts, DRIFT is able let firmware to trace itself. This self-tracing approach minimizes interference from the workstation, significantly boosting fuzzing performance.

We designed DRIFT to be highly flexible, accommodating a number of hardware resource limitations. When applied to new firmware, DRIFT discovered three previously unknown bugs that were not identified by existing binary fuzzing techniques. Furthermore, DRIFT outperforms all state-of-the-art binary firmware fuzzers in terms of speed and fidelity, trailing only SHiFT, an approach that requires source code.

I. Introduction

Fuzzing, a highly successful dynamic analysis approach for desktop programs, faces challenges when applied to embedded systems. This is primarily because of embedded system's diverse hardware configurations and, more importantly, constrained hardware. As a result, researchers have customized and developed a number of fuzzing approaches [19], [9], [17], [15], [22], [7], [6] for firmware, its software component.

The challenges of firmware fuzzing can be summarized as follows: *physical embedded systems are too resource-constrained to host fuzzing infrastructure, while emulated systems fail to reproduce faithful hardware behavior*. Emulation-based approaches, which avoid dependence on physical platforms, must emulate both the processor and its peripherals. While processor emulation is relatively mature thanks to decades of development, peripheral emulation is far more difficult due to the enormous diversity of hardware. Consequently, prior work has proposed heuristics to approximate generalized peripheral behavior and thereby enable fuzzing [3], [9], [19]. However, these approaches often suffer from low fidelity [8] and high false-positive rates [17]. In contrast, hardware-based approaches [21], [15], [17], [7], [14] avoid these drawbacks. Yet, because embedded platforms lack computational power, their workload must be minimized. As a result, these methods depend on intensive communication over I/O ports, which in turn becomes a major performance bottleneck [23], [17], [12].

While the main focus of the research community has been on improving the efficiency of firmware fuzzing, other work has approached the problem from an orthogonal angle: increasing bug detection capability. This question was first raised by Wycinwyc [18], which observed that in embedded systems—whether physical or emulated—a significant portion of triggered bugs were detected only much later, or not at all. To mitigate this issue, simple heuristics have been employed, such as timeouts [7], crash or interrupts [14]. More recently, SHiFT introduced the use of sanitizers, which substantially improved bug detection capability [17], albeit with the drawback of requiring access to firmware source code.

This work proposes a novel design that addresses the performance bottleneck of hardware-based approaches, while simultaneously leveraging interrupt-based mechanisms for bug detection. In comparison with recent related work, CO3 [15] and SHiFT [17] achieve state-of-the-art performance, but require access to source code. This limitation is particularly restrictive in firmware research, where a large portion of firmware is distributed in binary-only form. By contrast, uAFL [14], SAFIREFUZZ [21], and GDBFuzz [7] operate without source code. However, uAFL depends on a rarely available tracing unit—the ARM Embedded Trace Macrocell (ETM)—to extract coverage information, and still suffers from severe I/O bottlenecks that degrade performance. SAFIREFUZZ dynamically rewrites ARMv7-M firmware binaries for execution on ARM Cortex-A platforms, but this approach requires substantial manual effort and risks peripheral fidelity loss.

GDBFuzz, the work most closely related to ours, proposes the use of a debugging probe to algorithmically set breakpoints in order to infer code coverage of firmware running on physical hardware. However, the debugging probe has been shown to be cumbersome and disruptive to the real-time requirements on which firmware typically depends [15], [17]. Moreover, the breakpoint deployment algorithm in GDBFuzz is fundamentally limited. It distributes a small number of hardware breakpoints randomly and infers only *partial block* coverage through dominance analysis. As a result, it cannot provide *complete edge* coverage, which severely undermines

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00211

its fuzzing capability. As demonstrated in our evaluation (Section VI-B), this restriction in coverage significantly hampers GDBFuzz's effectiveness, rendering it inferior to both SHiFT and DRIFT.

This paper proposes using the Debug Monitor (DM) [1], a hardware feature available on nearly all ARM Cortex-M processors, to eliminate the cumbersome debugging protocol and achieve full edge coverage. To leverage DM for firmware fuzzing, we introduce Debug-based tRace Inference for Firmware Testing (DRIFT). The core insight of DRIFT is to implement the fuzzing infrastructure directly within the DM handler. This design removes the need for both firmware source code and external debugging probes. Because DM operates in close proximity to the CPU core, DRIFT achieves a significant performance advantage over probe-based approaches such as uAFL and GDBFuzz. Furthermore, with a carefully designed mechanism, we chain DM interrupts to allow the firmware to trace its own execution. Specifically, DRIFT first performs static analysis of the firmware binary to extract its control-flow (CF) information. This information is then embedded into the DM handler, enabling self-tracing of the firmware and supporting complete edge coverage during fuzzing.

Putting DRIFT to work requires simple static analysis and binary rewriting. Users can select their preferred greybox fuzzer (e.g., AFL or AFL++) to initiate the fuzzing campaign. Our experimental results demonstrate that, in terms of overall fuzzing performance, DRIFT outperforms all binary-only fuzzing approaches, including uAFL (the tracing-unit-based approach), P2IM, Fuzzware (the emulation-based approaches), and GDBFuzz (the halt-mode-debug-based approach), while trailing only SHiFT, the source-code-based approach.

In summary, we make the following contributions:

- We introduce a novel binary firmware fuzzing approach DRIFT, featuring in the utilization of DM, a widely-available hardware feature.
- DRIFT significantly enhances binary firmware fuzzing performance, surpassing all state-of-the-art methods, trailing only SHiFT, an approach that requires the firmware source code to be available.
- Leveraging the unique characteristics of DRIFT, we successfully identified three new bugs in two real-world firmware samples that were undetected by other binary firmware testing approaches.

II. Background

In this section, we begin by providing background information on microcontrollers (MCUs), which serve as the primary hardware platform for firmware. Given the resource-constrained nature of MCUs, prior research has explored various hardware features to recover coverage information during the fuzz testing of firmware executed on these devices. We then systematically analyze the existing mechanisms, outlining their advantages and limitations. Finally, we introduce the DM feature as a novel contribution to the field of firmware fuzzing.

A. MCU Architecture

MCUs are compact, self-contained computers that operate within constraints of computing power, hardware features, and data storage. Despite these limitations, they excel in performing mission-critical tasks, meeting real-time constraints, and operating efficiently in battery-powered scenarios. Typically, an MCU integrates peripherals into the processor's address space via memory-mapped I/O. These peripherals provide essential functionalities that firmware depends on. For instance, an MCU is commonly equipped with timers and watchdogs for time-sensitive operations, USART and I2C for data communication, and a Cyclic Redundancy Check (CRC) accelerator to ensure communication correctness and integrity. DRIFT leverages these essential hardware components, which are commonly available on MCUs, to create a functional and efficient fuzzing environment for firmware testing.

Previous work [18] has classified embedded systems based on the type of operating system (OS) they run. In this paper, we focus on Type-II and Type-III embedded systems, which are real-time OS-based and bare-metal designs, respectively. Type-I embedded systems, which run general-purpose operating systems, already have fully functional fuzzing environments available, and are therefore outside the scope.

B. Recover Coverage for on-device Firmware

ARM Cortex-M processors are by far the most widely adopted architecture for MCUs [10]. As discussed in the introduction, MCUs lack the computational resources to host an entire fuzzing infrastructure. Among the most critical components of this infrastructure is coverage feedback. While such feedback is often taken for granted in workstation fuzzers, it becomes a major challenge in MCU environments. On desktop systems, the operating system provides rich mechanisms that allow fuzzers to easily extract detailed information from target programs. In contrast, MCUs operate on fundamentally different hardware with far fewer resources. Furthermore, the physical connection between a desktop host and the MCU is typically constrained by the limited connectivity of the MCU, which falls far short of workstation-grade interfaces.

As a result, researchers have to utilize debugging features on the Cortex-M processors to extract coverage information, and feed it back to the fuzzer running on the desktop. The most notable debugging features are the halt-mode debugging and the tracing unit.

Halt-Mode Debug (e.g., JTAG or SWD): This is the most widely-used and straightforward mechanism in existing work. It relies on the JTAG or SWD protocols supported by the MCU. In essence, these protocols expose the MCU as a GDB server, allowing a fuzzer to attach through a GDB client over a physical channel (e.g., serial or USB). Once the connection is established, the GDB client can issue commands to halt the MCU processor and perform various debugging operations.

[1] https://docs.zephyrproject.org/latest/services/debugging/debugmon.html

	Cortex-M0	Cortex-M1	Cortex-M3	Cortex-M4	Cortex-M7	Cortex-M23	Cortex-M33	Cortex-M55
ETM	✗	✗	⍻	⍻	⍻	⍻	⍻	⍻
DM	✗	✗	✓	✓	✓	✓	✓	✓

✗: unavailable, ✓: available, ⍻: optional

TABLE I: Comparison of availability between ARM ETM and DM.

Vendor	**Series**	**W/ ETM**	**W/O ETM**	Total
STM	Mainstream	117	325	442
	Wireless	7	58	65
	High Performance	451	0	451
	Ultra Low Power	236	212	448
NXP	K32	2	16	18
	LPC	348	421	769
	MCX	8	56	64
	MK	152	433	585
Total		1321	1521	2842

TABLE II: ETM's availability of STM32 and NXP 32-bit Microcontrollers

Since halt-mode debug is the primary mechanism used by firmware developers during development, it is supported by virtually all MCU vendors. It provides a rich set of operations, including: 1) setting breakpoints, 2) single-stepping through instructions, 3) examining and modifying CPU registers, and 4) reading from or writing to memory locations.

While halt-mode debug is the most popular mechanism adopted by several firmware fuzzers (e.g.,[23], [6], [7]), its limitations are also well-recognized. The primary bottleneck lies in the low-bandwidth physical channel available on most MCUs, which makes communication with the workstation sluggish. This slowness severely undermines fuzzing performance, as firmware often depends on real-time operations that the GDB protocol struggles to sustain. Consequently, some works attempt to accelerate the debug channel[12], while others sacrifice coverage completeness in exchange for speed [7].

Tracing Unit (e.g., the ETM): Compared with the halt-mode debugging, this tracing unit provides a much more high-end and powerful functionality: real-time tracing. First used by uAFL for fuzzing, this on-chip unit, captures firmware execution in real-time, and generates a continuous data stream. The high volume of this data stream necessitates the use of an expensive high-end debug probe [2] for connectivity. Once the data stream is decoded on the workstation, users can recover code coverage information to facilitate fuzzing.

The primary drawback of ETM-based fuzzing is its limited availability. To demonstrate this, we surveyed the popular ARM Cortex-M processors, as shown in Table I to compare the availability between ETM and DM that DRIFT relies on. According to this table, although both DM and ETM are not available on Cortex-M0 and Cortex-M1 series, DM is available on all the other series while ETM is optional. Therefore, it is up to the MCU vendor to decide whether to include ETM in their products.

To minimize cost, MCU vendors typically only incorporate the ETM in their high-end MCU series. To better understand ETM availability, we surveyed the datasheets for all ARM-based MCUs of two prominent MCU vendors, NXP and STM. This survey covered datasheets for 2842 32-bit MCUs. We considered an MCU to have ETM if the datasheet: 1) Highlighted ETM as part of the MCU's features, 2) Depicted ETM in the MCU's block diagram, or 3) Mentioned ETM in a (sub)section. The result, summarized in Table II, reveals that more than half of the surveyed MCUs lack ETM support. This lack of availability makes ETM-based solutions impractical for widespread adoption.

The second disadvantage of using ETM for fuzzing is also the speed: since it captures the whole trace of the firmware execution on the MCU, it transmits verbose information. As a result, filtering the transmitted data becomes necessary. However, the speed is still undesirable even after filtering was applied [14].

Monitor-Mode Debug: As we can see from these two mechanisms, both of them have apparent downsides for firmware debugging and fuzzing. From the ARM Cortex-M processor's perspective, to address the drawbacks of halt-mode debugging (i.e., slow operation and disruption of real-time operations), it provides an alternative debugging mechanism: debug monitor Mode. This processor's feature utilizes a dedicated exception handler, the DM exception, within the Cortex-M processor [16], [5]. When it is enabled, hitting a breakpoint no longer halts the processor core. Instead, it triggers the DM interrupt, allowing a user-defined Interrupt Service Routine (ISR) to handle the event. The ISR associated with the DM exception provides a controlled environment for performing debugging operations. Users can program the routine to replicate the functionalities of halt-mode debugging without pausing the processor's core operations.

Compared with the server–client structure of halt-mode debug, DM offers the potential for a substantial performance boost in fuzzing due to its close integration with the processor core. This improvement allows coverage recovery algorithms to be designed without compromise. Moreover, since DM raises interrupts internally, it avoids the need to establish a GDB connection with the workstation. As a result, no external debug probe or debugging protocol is required. Instead, a simple serial port suffices for data communication, eliminating the reliance on facilities such as a GDB server or hardware debug probes. We build DRIFT on top of this DM-based debugging capability.

III. Related Work

A. *Emulation-based Firmware Fuzzing*

Emulation-based firmware fuzzing has attracted significant attention in recent years. Its primary strengths are hardware in-

[2] https://shop-us.segger.com/product-category/debug-probes/jtrace/

dependence and scalability. This approach can be categorized by the layer of the system it attempts to emulate.

Notably, P2IM [9], Fuzzware [19], and PRETENDER [11] focus on emulating the underlying hardware. Each introduces distinct heuristics to infer and replicate hardware behavior. However, the design of these heuristics involves an inherent tradeoff. On one hand, they aim to approximate the behavior of real hardware as closely as possible; on the other, they must maximize code coverage to make fuzzing effective. These two objectives are often in fundamental tension with one another.

Meanwhile, HALucinator [4], Para-rehosting [13], and SAFIREFUZZ [21] emulate the hardware abstraction layer (HAL) interface, thereby abstracting away the firmware's direct peripheral accesses. This approach improves fuzzing performance but comes with significant limitations. It depends heavily on error-prone binary function matching and, by design, cannot expose bugs that lie below the HAL—a substantial portion of many firmware codebases.

SAFIREFUZZ [21] extends the HALucinator concept by migrating code above the HAL from ARM Cortex-M to ARM Cortex-A, leveraging the fact that both belong to the same instruction set family. While this migration improves performance relative to HALucinator, it also inherits all of its fundamental drawbacks.

B. Hardware-based Firmware Fuzzing

Avatar [23] pioneered the hardware-in-the-loop design, in which on-device firmware communicates register-level states to the fuzzer through a debugging protocol. However, it was quickly recognized that the debug connection constituted a major performance bottleneck.

Subsequent work, such as Inception [6] and SURROGATES [12], attempted to overcome this limitation by accelerating communication with customized FPGA hardware. While these solutions improved throughput, their high degree of specialization and associated cost severely limited their practicality and broader adoption.

Moving forward, researchers observed that exchanging register states via a debugging protocol is both expensive and unnecessary for fuzzing. uAFL [14] addressed this by leveraging the ARM ETM to collect execution traces. However, as discussed in Section II-B, ETM is far less available than DM or halt-mode debug. It also introduces heavy computation and communication overhead, leading to significantly reduced performance.

More recent approaches, such as SHiFT [17] and CO3 [15], employ compile-time instrumentation to gather program states. Rather than relying on standard debugging protocols, they define lightweight custom protocols that transmit only the information relevant to fuzzing. This design achieves state-of-the-art performance while still supporting the full firmware stack running on the MCU. The drawback, however, is that these approaches require access to source code, which is often unavailable in real-world scenarios.

In comparison, GDBFuzz [7] continues to rely on the debugging protocol. Instead of synchronizing at the register level, however, the workstation communicates with the firmware at the fuzzing iteration level, which significantly reduces communication overhead. The key idea is to randomly insert hardware breakpoints and observe which ones are triggered. From these observations, the tester can partially infer basic block coverage using graph dominance analysis [3].

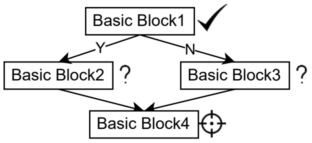


Fig. 1: the drawback of dominance analysis

This approach leverages the MCU's built-in debugging facilities without requiring source code access. Its limitation, however, is that it only provides partial basic block coverage. Complete edge coverage is generally more valuable for fuzzing, as it carries richer contextual information about execution paths [4] As an example, suppose that, in a simplistic control flow graph shown in Figure 1, a breakpoint is placed in basic block (BB) 4. Through dominance analysis, we are only able to infer that BB1 is guaranteed to be hit based on the fact that BB4 is hit. However, from BB1 to BB4, the control flow must travel through either BB2 or BB3. Dominance analysis is unable to recover this information, undermining the fuzzer's mutation capability. Furthermore, using the debugging protocol is still slow and breaks the real-time operations of the firmware.

Generally, DRIFT offers several benefits compared to GDBFuzz:

- Full recovery of the code coverage.
- No physical debug probe is required.
- No debugging protocol is used and no halting is needed. This ensured a smoother execution and faster speed.

C. Firmware Bug Detection Capability

The research challenges discussed above have largely defined the focus of the firmware fuzzing community. Wycinwyc [18] highlighted an additional problem: due to the absence of key hardware features (e.g., a memory management unit) and a full-fledged operating system, many classes of bugs that would be easily observable on a workstation remain hidden on MCUs.

As a result, firmware fuzzers often rely on coarse indicators such as hangs [7] or crashes [9] to infer that a bug has been triggered. To improve on that, uAFL [14], as an example, uses processor's fault handler as a signal, leveraging built-in error detection mechanisms to catch more failures.

The only approaches that go beyond these primitive methods are SHiFT and CO3, which embed sanitizers directly into the firmware. This enables much more fine-grained bug detection, but comes at the cost of requiring source code, making it impractical for binary-only scenarios.

[3] https://en.wikipedia.org/wiki/Dominator_(graph_theory)

[4] https://github.com/google/AFL/blob/master/docs/technical_details.txt#L58

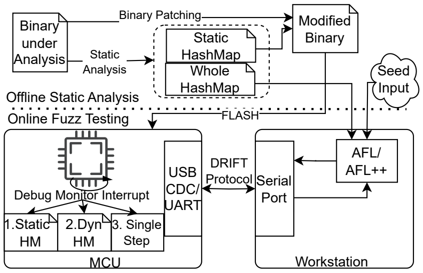


Fig. 2: System Block Diagram of DRIFT

CF Type	Action
1. Unconditional Jump	Place a breakpoint at the jump target
2. Conditional Jump	Place two breakpoints at both branches
3. Indirect Jump	Single step the indirect jump instruction
4. Direct Call	1. Place a breakpoint at the return address 2. Push the return address to shadow stack 3. Place a breakpoint at the start of callee
5. Indirect Call	1. Place a breakpoint at the return address 2. Push the return address to shadow stack 3. Single Step the call instruction
6. Hitting return address	1. Pop the return address from shadow stack 2. If hitting the end address, reset

TABLE III: Interrupt-based Actions for different control flows

IV. DRIFT Design

A. Workflow Overview

We present an overview of DRIFT in Figure 2. At a high level, DRIFT takes the binary-under-analysis as input. Through a simple control flow (CF) static binary analysis (detailed in Section IV-B), we generate key-value pairs containing the necessary information about how to collect code coverage from the binary. Due to the tight hardware budget on the MCU, only a limited number of pairs are stored in a static hashmap, which is subsequently embedded into the binary. All key-value pairs are provided to the fuzzer for use at runtime. Finally, the binary is further flashed onto the board for testing.

During online testing, we establish a connection with the communication channel used by the firmware and link it to the fuzzer. The fuzzer initiates a fuzzing campaign by sending an input to the MCU. The firmware, in return, operates in one of three action modes (illustrated in the MCU block in Figure 2 and detailed in Section IV-D) to collect code coverage information during its execution. After completing execution, the firmware transmits the error status and the collected coverage information back to the fuzzer, thereby concluding a fuzzing iteration.

B. DRIFT Core Design

The main focus of DRIFT is to utilize DM to recover coverage information. To do that, given a firmware binary, we need to first enumerate all its control-flow (CF) instructions. Then, for each identified CF instruction, we utilize the DM interrupt to dynamically recover its coverage information. For example, when DRIFT statically identifies a conditional branch at address `Addr1`. Through static analysis, DRIFT will know its only two targets are at `Addr2` and `Addr3`. At runtime, when the interrupt is triggered `Addr1`, DRIFT will place two DRIFT breakpoints at `Addr2` and `Addr3`. Through knowing which breakpoint is hit next, DRIFT is able to recover this coverage information.

Next we expand the same idea to all possible CF instruction. We list all of them in Table III. As it turns out, only two hardware breakpoints are required to recover the control flow (CF) for all CF instructions, except call/link instructions. To illustrate: For conditional branches, two hardware breakpoints are needed to monitor both potential targets. For unconditional branches, only one breakpoint is required at the single target address. For indirect branches (e.g., the `bx` instruction), no breakpoints are needed. Instead, we leverage the single-stepping capability of the DM to step through that instruction and record its target.

Handling call/link instructions involves more complexity because they not only trigger control flow (CF) transfers, but also modify the call stack. Thus, DRIFT is required to memorize where to return to when a function exits. To manage this, DRIFT needs a shadow call stack and an additional hardware breakpoint: The shadow call stack maintains the return addresses of on-stack functions. This additional hardware breakpoint is dynamically updated to the value at the top of the shadow call stack (i.e., the return address of the current function). This setup ensures that whenever a function returns, a breakpoint is placed at the return address, enabling the caller to resume its CF collection.

In summary, DRIFT requires three hardware breakpoints. According to our survey, all MCUs with DM mode provide at least four hardware breakpoints, thus DRIFT is applicable to all of them.

C. Hashmap Generation

DRIFT's core design reveals that, the DM interrupt essentially is embedded with a lookup table for each CF instruction. The key to the lookup table is each CF's instruction's address, the value of the entry is where to put the breakpoints. Thus, each entry can be generalized as following:

- (1 byte) the Type information which corresponds to the **CF Type** in Table III.
- (2 byte) first breakpoint to be placed, can be empty, which is denoted by `0x00`.
- (2 byte) second breakpoint, same as above.

Since this look-up operation occurs frequently during a fuzzing iteration (i.e., at the basic block level), it is crucial to use an efficient data structure for faster access. To achieve this, we choose the hashmap. Since we already know all the entries beforehand, when constructing the hashmap, we try different hash parameters (e.g., hash seeds and hashmap capacity) and pre-arrange the layout of the hashmap for improved space and time efficiency.

Despite our efforts to optimize the hashmap's space and time efficiency by pre-arranging its layout, a major bottleneck remains: a fuzzing iteration usually just exercises a small portion of the the hashmap. This issue arises because the firmware binary includes the entire software stack. As a result, hosting all of the code in hashmap is wasteful for the MCU's constrained memory as well as compute. We thus need a more flexible way to decide what to host in the hashmap by letting the user define the code of interest to test.

D. Enabling User-defined Hashmap

To enable this flexibility, we introduce two additional action modes in the design of the DM handler routine. Together, these three action modes are depicted in the MCU block of Figure 2.

The new hashmap generation starts with the user specifying the entry point to the code that needs testing. Then, the static analysis recursively explores through the call graph and generates entries for all traversed functions. However, there is a well-known limitation of binary static analysis – namely, its inability to accurately resolve indirect call targets or even conditional branch targets. The former causes the indirectly called functions to be not traversed and, consequently, missing from the hashmap; while the latter causes the the static analysis result to be wrong.

To dynamically amend the static hashmap, we introduce a dynamic hashmap. This dynamic hashmap addresses the two issues above: 1. when the MCU encounters a basic block that is not covered by the static hashmap, the MCU will request its entry from the workstation on-the-fly. The workstation, with the complete static analysis result, is able to provide any requested basic block information. 2. when the static analysis's result is wrong, which is detected when the current breakpoint trigger does not belong to any of the two addresses that were set by the previous breakpoint, the MCU is able to mark the previous breakpoint as *bad*.

When the MCU encounters a *bad* breakpoint, or the MCU runs out of space to request more entry, DRIFT will switch to the third mode, i.e., the single-step mode. In this mode, the MCU simply single-steps the each instruction and triggers DM interrupt for each. The interrupt will simply record the CF flow as it goes. More specifically, it checks if each instruction is a CF instruction. If it is a CF instruction, its target is recorded and added to the code coverage. If it is not, the system skips to the next instruction. Thanks to ARM's RISC-based design, determining whether an instruction is a CF instruction is straightforward [5]. For instance, to identify if a current instruction is a 16-bit conditional branch, we only need to check if the first 4 bits are `1101`, or if the first 5 bits are `11100`. If either is true, it is a conditional branch instruction.

In summary, DRIFT searches for key-value entries from both the static hashmap on FLASH and the dynamic hashmap on RAM to guide the handling of each encountered breakpoint. If an entry is bad or missing, it falls to single-step. This design makes DRIFT's memory requirements highly flexible: with a limited memory budget, no entries need to be hosted and DRIFT just single-step; with an abundant memory budget, entries can be preloaded to accelerate performance.

E. Creating a Functional Fuzzing Environment

With the ability to trace execution and recover code coverage for firmware as described above, we now discuss how to transmit this critical information back to the fuzzer on the workstation to enable a fully functional fuzzing process.

Coverage Feedback To provide coverage feedback to the fuzzer, DRIFT supports two types: bitmap and trace. Bitmap is an efficient construct widely adopted by popular openbox fuzzers, such as AFL. In this type, each encountered edge is mapped, via a hash function, to an entry in a bitmap. The key is the edge hash, and the value is the number of times the edge has been executed. SHiFT [17] uses this construct but optimizes it by transmitting only non-zero entries instead of the entire 64KB bitmap, based on the observation that most entries remain empty. DRIFT incorporates a similar design to improve efficiency. However, bitmaps inherently suffer from hash collisions, which can lead to the loss of coverage information. This issue is particularly pronounced on MCUs, as they typically cannot afford the use of computationally expensive hash functions like those employed on workstations.

To address this limitation, DRIFT also supports trace as a form of coverage feedback. Specifically, we draw inspiration from the ETM, and use a similar encoding strategy to represent the entire trace. A single bit is used to encode the destination of a conditional branch, and eight bits are used to encode the target of an indirect branch. Since the start and end addresses are specified by the user, the entire trace can be reconstructed from this encoding in a straightforward manner. The fuzzer can then apply a more computationally expensive hash function on the workstation to reconstruct the bitmap. Experimental results in Section VI-B demonstrate that while this trace mode offers similar functionality to ETM-based fuzzers, it is significantly faster and leaner.

Error Handling In addition to code coverage, the fuzzer requires information about triggered errors as feedback, as these signify the detection of real bugs. To achieve this, we modify the error handler (e.g., the hard fault handler) within the firmware to send a message whenever an error is triggered.

Moreover, hitting the hard fault handler typically indicates that the firmware has entered an irreversibly corrupt state. To ensure the firmware can recover from such a state automatically, we utilize a watchdog timer. The watchdog timer is a basic peripheral available even on the most low-end MCUs (e.g., STM32L0 series). It resets the board automatically if a timeout exceeds a preset threshold. This functionality is critical to maintaining the availability of the firmware, even in the event of a fatal crash. In our implementation, we also leverage this functionality to ensure the firmware remains operational during testing, even if the testing causes it to crash.

V. Implementation

A. MCU Runtime

DRIFT's design is not dependent on any RTOS like SHiFT or CO3 do. Therefore, it has better support for baremetal embedded devices. DRIFT interacts with the peripherals through

[5] https://developer.arm.com/documentation/ddi0403/latest/

the vendor-specific HAL. Thus, to support other MCUs within the same vendor, one can just copy-paste the source code to the target compilation environment and compile to generate the binary code; to support the MCUs of different vendors, besides copy-pasting, one should replace the vendor-specific HAL too. As a result, the MCU runtime comprises 945 lines of C code and the optional 132 lines of murmur hash implementation when the user choose not to use the hardware hash.

B. Static Analysis and Hashmap

As part of GDBFuzz's implementation, it has a Ghidra component to linearly scan the firmware to identify all its CF instructions. Expanding on this basic functionality, we further analyze all identified CF instructions to categorize them according to DRIFT's design (e.g., to check if they are conditional jump or indirect jump), identify their jump targets, and subsequently generate the static and global hashmap in the form of C code with a balanced trade-off between space and computation efficiency. The static hashmap is used together with other DRIFT's MCU runtime, to generate the final firmware binary. The global hashmap is subsequently used to generate the fuzzer running on the workstation.

C. Binary Patching

After obtaining the binary code, we implant it into the target firmware by first compiling the DRIFT runtime as a static library with position-independent code and then appending its symbols into the binary using Ghidra's patching functionality. We subsequently locate the DM handler and the fault handler in the binary's interrupt vector table and overwrite their entries so that they point to the injected functions, ensuring that the firmware invokes the implanted handlers whenever a breakpoint is triggered or a fault is detected.

Additionally, in order to initialze DRIFT's runtime (e.g., place a breakpoint at the `entry address` and force the binary to execute in DM mode) we effectively prepare a trampoline for the `main` function. Specifically, we add another function which does the aforemention initialization and then calls `main`. We then only need to overwrite `bl main` instruction inside the `Reset_Handler` to call a trampoline.

Furthermore, users might also be interested in testing the privileged code (e.g., the ISRs). Different from a normal function call, a given interrupt can be raised at any time during the firmware execution. DRIFT will not be aware when such an interrupt is raised. To support the potentially interesting interrupts, we also use the trampoline. Specifically, given an ISR (e.g., `USART2_IRQHandler`) that needs tracking, we simply overwrite its interrupt vector table's entry to point to the following trampoline:

```
DRIFT_USART2_IRQHandler():
    uintptr b1 = read_breakpoint(1);
    uintptr b2 = read_breakpoint(2);
    uintptr b3 = read_breakpoint(3);
    write_breakpoint(1,USART2_IRQHandler);
    call USART2_IRQHandler();
    write_breakpoint(1,b1);
    write_breakpoint(2,b2);
    write_breakpoint(3,b3);
```

As one can see from the code above, in order to make sure that the interrupt does not destroy the breakpoint context of the function that is being interrupted, we save the breakpoint context into the system stack. In doing so, when the interrupt finishes, the interrupted function can continue its CF recovery normally. Moreover, in order to make sure that the interrupt does not destroy DRIFT's internal state, we disable all interrupts when the DM handler was executed and re-enable before it exits.

D. AFL Proxy

Once the modified binary is flashed to the MCU, as shown in Figure 2, depending on how the data is consumed on the board, DRIFT needs to set up the same input channel used by the firmware. This is done by setting up an afl proxy [6]. From the perspective of the fuzzer, this afl proxy is the target program under test. What this proxy does under the hood is that it serves as an intermediary between the fuzzer and the MCU. It first receives input from the fuzzer and writes to the input channel used by the firmware. Subsequently, it also receives the feedback from the MCU to reconstruct the bitmap needed for fuzzer's input mutation.

VI. Evaluation

In this section, we systematically evaluation DRIFT to answer the following research questions:

- RQ1: In order to put DRIFT to work, what are the overhead that an user should expect?
- RQ2: Compare with other SoTA hardware-based solutions, how does DRIFT's fuzzing performance in terms of speed and code coverage?
- RQ3: Due to DRIFT's unique hardware-based error detection mechanism, what types of bugs can DRIFT detect?
- RQ4: What are the bug-detection capabilities of DRIFT compared with the existing SoTA? Is DRIFT able to find new bugs that other solutions cannot find?

We conducted experiments on a workstation equipped with Intel Core i9-12900H processor and 32 GB RAM. In terms of the MCU boards, as listed in Table IV, we ported DRIFT to MCUs from STM32 [7], Microchip [8], and NXP [9]. We chose these vendors because they are among the most prominent ARM MCU providers [2].

We chose uAFL and GDBFuzz as the primary comparison targets, since similar to DRIFT, they are both hardware-based binary firmware fuzzing approaches. When it comes to uAFL,

[6] https://github.com/AFLplusplus/AFLplusplus/blob/stable/utils/afl_proxy/afl-proxy.c

[7] STM32 MCU Selector, https://www.st.com/content/st_com/en/stm32-mcu-product-selector.html

[8] Microchip Product Selection Tool, https://www.microchip.com/en-us/products/selection-tools

[9] NXP Product Selector, https://www.nxp.com/products/product-selector:PRODUCT-SELECTOR.

Vendor	MCU	CPU	Connectivity	FLASH	RAM	HASH	Watchdog
STM32	*F103RB*	Cortex-M3 @ 36 MHz	UART @ 7.5 Mbauds	128KB	20KB	Hardware CRC	x2
STM32	*F429ZI*	Cortex-M4 @ 180 MHz	UART @ 7.5 Mbauds	2MB	260KB	Hardware CRC	x2
STM32	*L4R5ZI*	Cortex-M4 @ 120 MHz	USB 2.0 @ 12Mbps	2MB	640KB	Hardware CRC	x2
STM32	*L4S5VIT6*	Cortex-M4 @ 120 MHz	USB 2.0 @ 12Mbps	2MB	640KB	Hardware SHA-256	x2
STM32	*H743ZI*	Cortex-M7 @ 480 MHz	USB 2.0 @ 12Mbps	2MB	1024KB	Hardware CRC	x2
NXP	*K66F*	Cortex-M4 @ 180 MHz	USB 2.0 @ 12Mbps	2MB	256KB	Hardware SHA-1	x1
Microchip	*SAMD51*	Cortex-M4 @ 120 MHz	USB 2.0 @ 12Mbps	512KB	256KB	Hardware SHA-1	x1

TABLE IV: List of MCU platforms

however, due to its dependency on the expensive high-end debug probe [10], dependency on specific OpenOCD version, and unreliable setup, we ran into a similar problem as described in GDBFuzz [7] when trying to reproduce uAFL's experiment. As a result, we are only able to use the result reported from its original paper (i.e., for the `Console` and `Fibonacci` firmware). Besides these on-device binary fuzzing approaches, to figure out where DRIFT situates in firmware fuzzing in general, we also compare with other state-of-the-art firmware fuzzers. In particular, we compare with SHiFT, a source-code-dependent fuzzer that runs the firmware on the MCU and demonstrates state-of-the-art performance, and Fuzzware, a cutting-edge emulation-based firmware fuzzer.

When it comes to choosing the benchmark firmware, we surveyed the availability, practicability and variety of the firmware evaluated in the literature on firmware testing. We found a balanced collection of firmware samples from P2IM, SHiFT, CO3 since they offered a well-documented firmware samples, with varying degrees of complexity. For a fair comparison, we also adopt the same seed inputs. DRIFT managed to find three new bugs in two new firmware samples, thus we also added them to the benchmark. They are `oled`, a OLED-based display driver and `UVC-VGA`, a webcam adapter which translates webcam signal to the computer. To demonstrate this benchmark's variety, we present three metrics for these firmware, namely, number of basic blocks, number of hardware peripherals that each firmware accesses, and the number of indirect calls each firmware makes. The former two showcases the general complexity of the firmware with number of basic blocks being a general indicator of how complex the code is and the number of hardware peripherals gives a sense of the peripherals involves. We also present the number of indirect calls each firmware makes since this poses challenges to the static binary analysis of DRIFT. This challenge is especially true considering function pointers are pervasive in firmware. Fortunately, DRIFT handles such cases via switching to single-stepping.

A. RQ1: On-board Overhead

Applying DRIFT to the binary-under-analysis requires overwriting existing interrupt handlers and instructions by using trampolines. These newly added code increases the memory footprint for the given firmware. To understand the the overhead, we compiled DRIFT for all the firmware samples across different vendors. The result is shown in Figure 3.

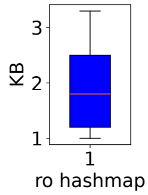
As shown in Table 3a, we break down the memory overhead into the trampolines, which are essential to the functionality of DRIFT, and the software hash components, which are configurable and vary from firmware to firmware. In terms of the trampolines, the code itself consists of 3.6KB throughout all the evaluated benchmarks. It further adds 1.1KB to the read-only section, which is primarily static strings, and only 0.1 KB to RAM which comprises the shadow stack and other global variables.

In terms of the software hash components, the code itself is 1.1KB. Note that the user can avoid this overhead by using the hardware hash if it is available on the MCU. The read-only section directly corresponds to the size of the static hashmap and heavily depends on the firmware. Thus, we drew a boxplot for the different sizes of the static hashmaps in Figure 3b. As it shows, the median of the static hashmap is 1.9KB across all the benchmarks. The RAM overhead is allocated for hosting the dynamic hashmap. We thus empirically set it to 0.5KB which we deem to be reasonable across all the tested MCUs.

In summary, DRIFT imposes minimal memory overhead on the MCU. The essential part supports tracing with single stepping and only requires 4.7KB of FLASH and 0.1KB of RAM. To make it work more efficiently by leveraging the hashmap, an additional 3KB of FLASH and 0.5KB are typically needed.

Mem	Seg	Trampolines	SW Hash*
FLASH	text	3.6 KB	1.1 KB
	ro	1.1 KB	1.9 KB#
RAM	data	0.1 KB	0.5 KB

(a) DRIFT memory overhead. * marks the part that varies depending on the configuration and firmware. # corresponds to the median static hashmap size.



(b) Static hashmap footprint

Fig. 3: DRIFT memory footprint.

B. RQ2: Fuzzing Performance

Speed One of DRIFT's main design goals is to boost fuzzing speed, which serves as a fundamental metric for evaluating fuzzing performance. We thus ran DRIFT with other works against all the collected firmware samples and evaluated their fuzzing speed.

As Table V shows, DRIFT maintains a clear speed advantage over the direct competitors (i.e., uAFL and GDBFuzz). When it comes to the outreaching competitors, DRIFT also has

[10] https://shop-us.segger.com/product-category/debug-probes/jtrace/

Ref	#	Firmware	Board	BB	HW acc	Ind Call	DRIFT			SHiFT			uAFL			Fuzware			GDBFuzz		
							[r/s]	TP	FP	[r/s]	TP	FP	[r/s]	TP	FP	[r/s]	TP	FP	[r/s]	TP	FP
P2IM [9]	1	PLC	H743	473	7	33	870	4	0	3100	4	0	n/a	n/a	n/a	33	4	2	67	4	0
	2	Console	H743	95	2	24	453	0	0	2800	0	0	2.3	0	0	35	0	0	48	0	0
SHiFT [17]	3	Synthetic	H743	64	3	29	1840	10	0	4800	13	0	n/a	n/a	n/a	87	0	10	35	2	0
	4	AT parser	SAMD51	89	3	9	103	0	0	276	0	0	n/a	n/a	n/a	54	0	2	55	0	1
	5	Commandline	K66F	48	4	36	87	0	0	233	0	0	n/a	n/a	n/a	332	0	1	248	0	4
CO3 [15]	6	CANopen	L4R5	104	5	23	63	0	0	144	3	0	n/a	n/a	n/a	46	0	0	12	0	1
	7	Bldc	F429	240	11	89	32	2	0	103	2	0	n/a	n/a	n/a	12	0	0	15	1	3
GDBFuzz [7]	8	Http	L4S5VIT6	166	4	11	78	0	0	198	0	0	n/a	n/a	n/a	35	0	0	26	0	0
DRIFT	9	Oled	F103	146	3	13	122	2	0	254	2	0	n/a	n/a	n/a	34	0	4	22	1	2
	10	UVC-VGA	F429	119	6	21	143	1	0	379	1	0	n/a	n/a	n/a	14	0	1	19	0	0

TABLE V: 24-hour fuzzing campaigns of DRIFT compared to the SoTAs. **BB**: Number of basic blocks in the firmware, **HW acc**: How many peripherals are accessed in the firmware, **Ind Call**: How many indirect calls exists in the firmware, **SU**: DRIFT SpeedUp (average), **TP**: True Positives (median), **FP**: False Positives (median), **NB**: No Bootstrap.

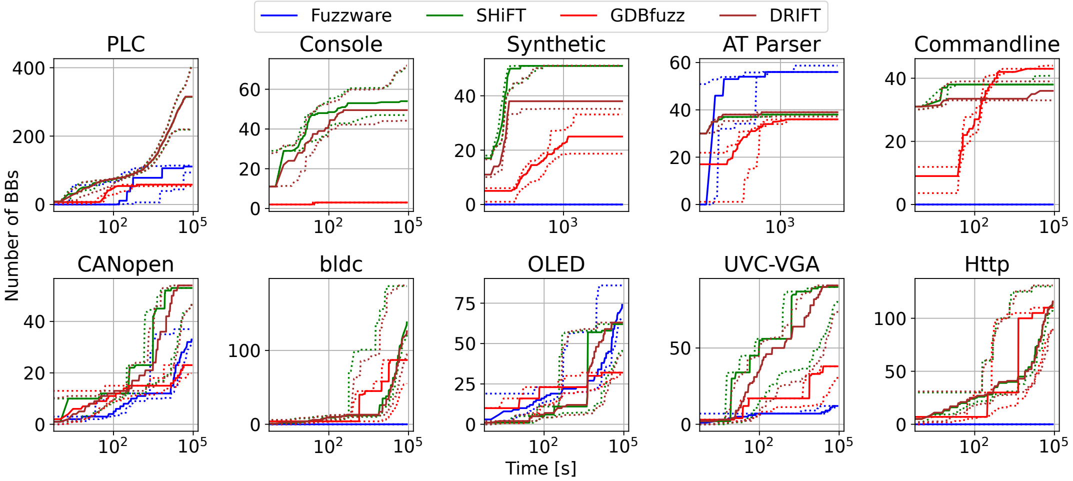


Fig. 4: Comparing basic block coverage. The solid lines are the median block coverage across all runs and the dotted lines represent the 95% confidence interval.

a clear speed advantage over the emulation-based approach, losing only to SHiFT, an source-code-instrumentation-based approach.

Specifically, when it comes to uAFL, DRIFT outperforms it by two orders of magnitude in the `console` firmware. As reported from uAFL [14], its low speed is mainly due to the high demand for trace communication and filtering. DRIFT also generally outperforms GDBFuzz by one to two orders of magnitude. GDBFuzz halts the processor to deploy breakpoints and engage the gdb protocol for every fuzzing run. This speed gap between the serial GDB communication and the CPU causes performance issue. In comparison, DRIFT uses interrupt to deploy breakpoints and does not halt the processor. The only exception comes from the `Commandline` where GDBFuzz is faster than both DRIFT and SHiFT. This is because `Commandline` contains simple logic (e.g., comparing the input characters with hard-coded ones and it is composed of about 40 basic blocks). As a result, all works saturated the firmware early in the fuzzing campaign. Consequently, GDBFuzz does not have to distribute more breakpoints while SHiFT and DRIFT were still paying for the overhead of the injected runtime. This eventually averaged out with GDBFuzz coming to the top. Such a phenomenon was not seen in other firmware samples.

Besides the hardware-based approaches, DRIFT also outperforms Fuzzware, the emulation-based approach by one order of magnitude. We attribute this to DRIFT's design goal of following a semihosting-based design. Unlike emulation-based approaches, semihosting-based approaches run as much fuzzing infrastructure as possible on the board. Doing so avoids the heavy lifting of the binary code and peripheral modeling which are required by the emulation. While this is true for all firmware samples, we observed one outlier which is `Commandline` where Fuzzware is significantly faster. We noticed that, this is due to the early failure caused by the Fuzzware emulation. As a result, Fuzzware never reached the entry point of the `Commandline` main logic. Since the failure is in the early stage of the firmware booting, Fuzzware ended up only exercising a small portion of the firmware, making each execution fast but not in a meaningful way.

When comparing with SHiFT, both SHiFT and DRIFT use similar mechanisms for collecting on-device code coverage. SHiFT uses software instrumentation to collect coverage, whereas DRIFT runs similar functionality inside the DM handler. As a result, the disadvantages of DRIFT are 1. for every hit breakpoint, DRIFT needs to switch to the DM interrupt. This involves a necessary context switch; whereas SHiFT directly embeds the instrumentation code inside the source code, avoiding this overhead. 2. besides the context switch, DRIFT also needs to perform a hashmap lookup to look for the execution plan for each hit breakpoint; whereas SHiFT just performs direct memory access without any lookup. As these two factors happen for each encountered basic block, the overhead eventually adds up and causes a speed disadvantage.

In summary, DRIFT maintains an apparent fuzzing speed advantage over the hardware-based binary approaches thanks to its proximity to the CPU core. However, it is still slower than the source-code-based approach due to the inevitable overhead incurred by performing the same functionality in source code versus binary code.

Code Coverage In the same experiment, we also ran each fuzzing campaign ten times for twenty-four hours to follow the best practice of evaluating code coverage [20]. While DRIFT and other works support edge coverage by design, GDBFuzz is only able to support the coarser-grained basic block coverage as feedback [7]. For a fair comparison, we chose basic block coverage as the feedback metric for all evaluated systems. We aggregate the results and plot the median basic block coverage and its 95% confidence interval in Figure 4. Additionally, we conducted Mann-Whitney U test on the sequence of each number of basic blocks, the calculated critical value meets satifies the significance level of $\alpha = 5\%$, showing that the collected data demonstrated statistical significance.

From the figure we can see that, DRIFT outperforms both GDBFuzz and Fuzzware by a considerable margin. Fuzzware, in particular, achieved greater coverage on `AT Parser` and `OLED`, however, we categorize this phenomenon to its forceful code coverage strategy in modeling the peripherals. As a result, the peripheral modeling yields unrealistic values which boosts up the coverage. In terms of GDBFuzz, it consistently underperforms due to its slow speed and partial basic block coverage.

When comparing SHiFT and DRIFT, these two systems have similar behaviors in `Synthetic`, `Console`, and `AT Parser`. This is because, compared with other works, the way both systems use to collect coverage is similar. On a high level, although the detailed timing and behavior might be different due to fuzzing randomness, SHiFT uses source code instrumentation to collect coverage while DRIFT runs the same logic in the form of an interrupt.

C. RQ3: Types of Bugs Detected

As stated in Section IV-E, DRIFT detects crashes and time-outs based on the fault handlers of the MCU. This bug detection mechanism is inherently much more coarse-grained and not as powerful, compared with the software-based sanitizers (e.g., the AddressSanitizer used by SHiFT). Since most of the existing firmware fuzzers utilize the emulator's rich mechanisms for bug-detecting purposes, less is known in terms of what can be expected when it comes to utilizing MCU's on-chip mechanisms to detect bugs. For example, GDBFuzz only uses hang to signify when a bug is detected. To bridge this gap, SHiFT proposed a `Synthetic` benchmark that enumerates thirteen different commonly seen bugs with simple triggering conditions. This benchmark also categorized each bug according to how it can be detected, i.e., if the bug can be detected through the 1. memory protection unit (MPU), 2. Cortex-M fault handlers, or 3. software sanitizers. We also use this benchmark to quantify how many bugs DRIFT can detect.

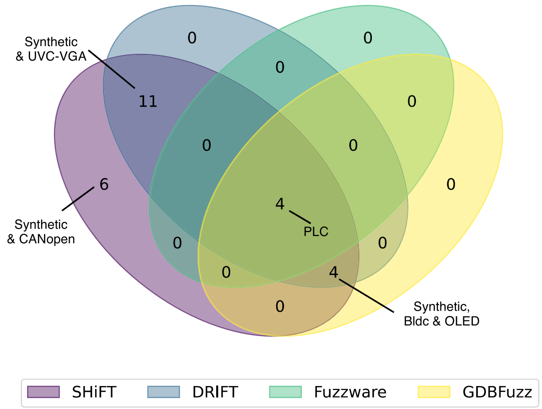


Fig. 5: Intersection of identified bugs by different works.

When running DRIFT on this benchmark, naturally, all the bugs marked detectable by Cortex-M fault handlers can also be detected by DRIFT. These bugs are divide-by-zero, invalid instruction, privilege violation, null pointer dereference, and unaligned access. DRIFT cannot detect MPU-specific bugs since DRIFT does not rely on the MPU. This includes segmentation fault. When it comes to the sanitizer-detected bugs, DRIFT also detects all bugs (except stack overflow) marked detectable by ASAN while failing to detect all UBSAN bugs. The ASAN-detected bugs are use-after-free, double-free, invalid-free, heap-overflow, and the UBSAN-detected bug is integer overflow. We categorized the fact that DRIFT is able to detect ASAN bugs to that, these detectable bugs use `libc` functions. Once the bugs are triggered, it leads to corruption inside the `libc` functions and was subsequently captured by hard fault.

D. RQ4:Bugs Detected

In the process of running all related works based on the benchmark, we also evaluated their bug detection capability against the existing bugs documented in the previous papers, as well as the new bugs that DRIFT found in the new firmware samples. The results are shown in Table V and Figure 5.

Thanks to DRIFT's efficient and high-fidelity fuzzing for firmware, after we applied it to the real-world firmware samples collected from the Internet, DRIFT was able to trigger three new bugs that other binary works did not find. The first bug is in the `oled` firmware that drives a oled display. It implements a library for easier drawing on a stm32-based screen. This bug arises due to a poor boundary check when writing to a global buffer, which put the firmware in a corrupted state and triggered a hard fault. Fuzzware failed to detect this because the firmware utilizes on-chip peripherals to interact with the display. However, Fuzzware misclassified the peripheral's registers, which led to an unrealistic program path. Meanwhile, GDBFuzz struggled with reaching that buggy part of the code due to its limited exploration capability, which was undermined by its partial feedback. A similar situation arises, when it comes to `UVC-VGA` firmware where DRIFT identified a heap overflow and a null pointer dereference. However, in this case, Fuzzware was stalled before it hit the buggy part of code. In both cases, the bugs are confirmed by the developers after we report to them responsibly.

When it comes to detecting the existing documented bugs, DRIFT cannot detect the undefined behaviors inside `CANopen` since it bears no noticeable behavior. Meanwhile, it is able to detect the other bugs that were made noticeable by triggering the fault handlers. Fuzzware, in comparison, struggled with reaching to the main processing loop of the firmware and triggering false bugs as seen in `Commandline` and `Bldc`. GDBFuzz, at the same time, primarily suffered from low speed and limited fuzzer mutation while having a better false positive rate compared with Fuzzware since it runs the firmware on the real MCU. We drew a venn diagram to visualize the true findings made by each work in 5. As one can see, SHiFT is able to detect all the true findings due to its sanitizer-based design, while DRIFT comes as a close second. GDBFuzz and Fuzzware, in comparison, suffers from their limitations and thereby are only able to discover a fraction of the findings, with GDBFuzz being relatively better since it uses the real hardware.

VII. Discussion

A. Architectural Limitations

DRIFT is built upon the DM mode provided by ARM Cortex-M, which is by far the most popular MCU processor family [1]. However, what DRIFT primarily depends on is the processor's ability to handle triggered breakpoints independently, without interference from the workstation. This capability, combined with the pre-knowledge of the firmware, makes DRIFT both lightweight and fast.

Although we do not support other MCU architectures out-of-the-box, it is not uncommon for other architectures to offer similar capabilities for minimally intrusive debugging. For example, RISC-V supports *minimally intrusive debugging*[11], which allows the execution of specific instructions during debugging without interference from the workstation. Similarly, PowerPC offers a *debug interrupt*[12], which triggers a debug exception when specific events occur. The design principles behind DRIFT provide a paradigmatic framework that can be adapted when porting to these architectures, a task we leave for future work.

B. Tracking the Interrupt

In the implementation, we discuss how DRIFT supports tracking interrupts. However, since MCUs strictly arbitrate interrupt handling based on priority (i.e., high-priority interrupts always preempt lower-priority ones), tracking high-priority interrupts (i.e., those with higher priority than the DM) becomes impossible. Although we could technically set the priority of the DM interrupt to the highest value, we recognize that this is not the intended use of the DM and could negatively affect the firmware's original behavior, thus impacting its fidelity. Note that high-priority interrupts are typically used for system services, while user-defined interrupts usually have lower priorities. This distinction makes our approach still valuable in most practical cases.

C. Inaccurate Static Analysis Result

Accurate static analysis results are fundamental to the DRIFT's design. However, static binary analysis remains an open research question and suffers from inaccuracy.

We argue that this is less of a problem for DRIFT since 1. ARM Cortex-M is a RISC-based architecture: unlike x86 whose instruction can be from 2 to 15 bytes longs, ARM can only be from 2 or 4 bytes. This significantly reduces the complexity of static analysis. 2. DRIFT does not involve complex binary data analysis; only primitive CF analysis is needed. This does not even involve chaining different CF together–only analyzing each individual CF instruction is needed. 3. For the identified inaccurate basic blocks, we have a design in place to automatically switch to single-step mode.

Given these, we acknowledge that DRIFT cannot handle extreme cases such as obfuscated code or even self-modifying code, which we leave to future work.

VIII. Conclusion

Based on a widely-available Arm Cortex-M feature, the DM, DRIFT designs and implements a simple yet powerful binary firmware fuzzing framework.

Based on the evluation comparing with on-device binary firmware fuzzing systems as well as the current state-of-the-art source-code-based and emulation-based approaches, DRIFT maintains a considerable advantage over the directly related binary firmware approaches and emulation-based approaches, losing only to the source-code approach.

IX. Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This research is sponsored in part by NSF Award 2031390, NSERC (RGPIN-2022-03325), and NCC (2024-1488).

[11] https://riscv.org/wp-content/uploads/2024/12/riscv-debug-release.pdf

[12] https://www.st.com/resource/en/user_manual/um0434-e200z3-powerpc-core-reference-manual-stmicroelectronics.pdf

References

[1] The current state of embedded development. https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf.

[2] Microcontroller Market Size, Share & Trends Report 2030. https://www.grandviewresearch.com/industry-analysis/microcontroller-market.

[3] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. Multifuzz: A multi-stream fuzzer for testing monolithic firmware. In *USENIX Security Symposium*, USENIX Security, 2024.

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

[5] Chris Coleman. Step-through debugging with no debugger on cortex-m. URL: https://interrupt.memfault.com/blog/cortex-m-debug-monitor.

[6] 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.

[7] 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.

[8] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, May 2021.

[9] 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.

[10] Barr Group. Embedded Systems Market Surveys, June 2016.

[11] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, Chaoyang District, Beijing, September 2019. USENIX Association. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[12] 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.

[13] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. URL: http://arxiv.org/abs/2107.12867, arXiv:2107.12867, doi:10.14722/ndss.2021.24308.

[14] 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, 2022.

[15] Changming Liu, Alejandro Mera, Engin Kirda, Meng Xu, and Long Lu. Co3: Concolic co-execution for firmware. In *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[16] Zephyr Project members and individual contributors. Cortex-m debug monitor – zephyr project documentation. https://docs.zephyrproject.org/latest/services/debugging/debugmon.html, 2024.

[17] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[18] 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. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf, doi:10.14722/ndss.2018.23166.

[19] 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.

[20] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993, Los Alamitos, CA, USA, May 2024. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00137, doi:10.1109/SP54263.2024.00137.

[21] 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.

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

[23] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security analysis of embedded systems firmwares. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society.