ARTIFACT EVALUATED
USENIX
AVAILABLE

FirmReBugger: A Benchmark Framework for Monolithic Firmware Fuzzers

Mathew Duong[1], Michael Chesser[1], Guy Farrelly[1], Surya Nepal[2], and Damith C. Ranasinghe[1]

[1]*University of Adelaide*
[2]*Data61 CSIRO*

Abstract

Monolithic Firmware is widespread. Unsurprisingly, fuzz testing firmware is an active research field with new advances addressing the *unique* challenges in the domain. However, understanding and evaluating improvements by deriving metrics such as code coverage and unique crashes are problematic, leading to a desire for a reliable bug-based benchmark. To address the need, we design and build FIRMREBUGGER, a holistic framework for fairly assessing monolithic firmware fuzzers with a realistic, diverse, bug-based benchmark.

FIRMREBUGGER proposes using bug oracles—C syntax expressions of bug descriptors—with an *interpreter* to automate analysis and *accurately* report on bugs discovered, discriminating between states of *detected*, *triggered*, *reached* and *not reached*. Importantly, our idea of benchmarking does not modify the target binary and simply *replays* fuzzing seeds to *isolate* the benchmark implementation from the fuzzer while providing a simple means to extend with new bug oracles.

Further, analyzing fuzzing roadblocks, we created FIRMBENCH, a set of diverse, real-world binary targets with 313 software bug oracles. Incorporating our analysis of roadblocks challenging monolithic firmware fuzzing, the bench provides for rapid evaluation of future advances. We implement FIRMREBUGGER in a *FuzzBench-for-Firmware* type service and use FIRMBENCH to evaluate 9 state-of-the art monolithic firmware fuzzers in the style of a reproducibility study, using a 10 CPU-year effort, to report our findings.

1 Introduction

The surge in embedded systems, from cube-satellites to smart door locks and autonomous vehicles, is driven by small, low cost, and low-power microcontrollers. The proliferation of embedded systems creates more targets, new attack vectors, opportunities, and incentives for adversaries [40]. Testing firmware to identify vulnerabilities prior to exploitation or device failure is critically important. For example, Hanna *et al.*'s [27] assessment of defibrillator firmware in a commercial product revealed critical vulnerabilities, including a buffer overflow that could allow remote code execution.

Unsurprisingly, embedded systems' rising prevalence and associated security concerns are driving advances in automated software testing methods like fuzzing. However, uncovering potential security vulnerabilities in firmware with fuzzing is challenged by hardware limitations, resource constraints, and the complexity of direct firmware interactions with hardware. Consequently, firmware fuzzing is an active topic of research interest where recent methods [9, 10, 19–21, 35, 44, 55, 56] re-host firmware using emulation on more resourceful computing platforms to bypass the resource limitations of embedded hardware. Emulation provides the ability to add sophisticated instrumentation to guide a fuzzer, facilitates introspection, and enables better detection and reproducibility of crashes whilst avoiding the need to synchronize hardware and emulation environments with *device-in-the-loop* methods. Despite the benefits of re-hosting, applying automated testing methods like fuzzing remains challenging, especially for monolithic firmware.

Monolithic firmware directly manages all interactions with hardware, operating without support or supervisory control of an operating system, and combines all essential software components into a single unified entity. Most notably, in contrast to traditional applications, monolithic firmware interacts directly with peripherals (such as modems, GPS units, and sensors) through Memory Mapped Input Output (MMIO) or Direct Memory Access (DMA) mechanisms and interrupts. These low-level interactions are complex and may be across a diverse group of peripherals with behaviors differing between manufacturers. This poses significant challenges for employing a de-facto, grey-box, mutation-based fuzzing method to drive re-hosted firmware to discover software bugs.

Advances in fuzz testing firmware aim to overcome the pertinent challenges to achieve scalable, efficient, and effective fuzzers by facilitating the firmware's interaction with a diverse group of peripheral devices across various different microcontroller designs employing a range of different instruction set architectures (ISAs). But, a key question with

regards to the evaluation of fuzzers, to support both the growth and rapid developments of new techniques is:

> How do we fairly assess improvements to the *bug finding* capabilities of monolithic firmware fuzzers?

1.1 Unique Problems Challenging Assessment

Assessing the performance of fuzzers, in general, relies on *empirical* means to evaluate their bug-discovering capabilities. Code coverage and unique crash counts are adopted from domains outside of firmware targets to compare if one firmware fuzzer is "better" than another [9,10,19–21,44,45,55,56]. Unfortunately, adopting these accepted performance measures to evaluate monolithic firmware fuzzers is not without its unique set of problems. We delve into these issues in detail in Section 3 and summarised them below:

- *Code Coverage*. The relationship of coverage achieved by a fuzzer to its bug-finding ability support it as a performance metric [24, 29, 31]. However, its use in evaluating monolithic firmware fuzzers can lead to misleading results. For example, a fuzzer exploiting a bug to reach otherwise unreachable code can inflate the coverage metric, giving the illusion of better performance—a scenario more commonly encountered in embedded systems (see Section 3.1).
- *Unique Crashes*. Unique crashes area more direct measure of the number of bugs found. But, implementing it accurately in the firmware domain is challenging. Differentiating and deduplicating crashes require capturing a bug's context. Common identifiers, such as coverage profiles and stack hashes are prone to variability in program behavior, making consistent and reliable crash classification difficult. The problem is further exacerbated in the firmware domain where program flow is more complex due to the interrupt driven nature of firmware (see Section 3.2).

A bug-based benchmark is recognized as an approach to alleviate the imperfect nature of coverage and problems with unique crash metrics. In general, bug-based benchmarks such as LAVA [16] with LAVA-M, Magma [28], UNIFUZZ [32], Google's FuzzBench [36], FixReverter [54], and FuzzProBench [39] are proposed for *non-firmware* targets. However, a benchmark for evaluating monolithic firmware fuzzers does not yet exist. Notably, recent method for designing a benchmark in FixReverter [54] require access to source code, which is often unavailable with firmware images. And the bug-oracle approach in Magma [28] integrates the benchmark into the fuzzing logic and leads to the reported leaky oracle problem [28]. In addition, given the developing nature of firmware fuzzing there is a lack of effective tooling for automating the triaging of bugs. The alternative—curating direct performance results such as time-to-find bugs—necessitates manual crash analysis and triaging for each fuzzer, which is both time-consuming and error-prone.

Consequently, we are motivated to design a bug-based benchmarking framework for monolithic firmware fuzzers. We consider the unique challenges posed in the domain and facilitate automated and fair evaluation of fuzzers. We summarize our efforts in the following sections.

1.2 Our Work

We analyze the problems with empirical performance evaluation for firmware images (Section 3) and propose FIRMREBUGGER, a new benchmark framework together with FIRMBENCH target sets—challenging, real-world, benchmark binaries with a diverse set of bugs. Importantly, we curate binary targets through a careful consideration of existing monolithic firmware fuzzing challenges we discuss in Section 5.1.

Benchmark Design Goals. First, we extend the collective wisdom from the design of benchmarks outside of monolithic firmware and curate a desirable set of goals for a firmware benchmark. Notably, recent fuzzing benchmark studies, such as those highlighted by Zhang et al. [54] and Hazimeh et al. [28], have identified necessary objectives for creating an *effective* and *reliable* fuzzing benchmark. Drawing inspiration from these insights, we present the following goals we deem essential in developing a comprehensive benchmarking framework. The benchmark:

- ***G1*** Should not affect the fuzzing process (the leaky oracle problem).
- ***G2*** Should provide clear indicators of bugs triggered.
- ***G3*** Should use relevant, real-world target programs.
- ***G4*** Should contain targets with realistic and relevant bugs.
- ***G5*** Should defend against overfitting of fuzzers to targets.

To address *G1*, we isolate the fuzzer from the benchmark triaging process. This ensures that the benchmark itself does not influence fuzzing outcomes, mitigating the leaky oracle problem. Specifically, FIRMREBUGGER uses *bug oracles*. We create bug expressions to formally describe the conditions to confirm a specific bug. These expressions are interpreted by replaying both crashing and non-crash inputs (fuzzing inputs) to determine whether a specific bug is reached, triggered, or detected. By this analysis, FIRMREBUGGER automatically computes essential bug-based performance metrics—*G2*.

To satisfy *G3* and *G4*, FIRMREBUGGER uses a curated set of binaries covering widely used real-world firmware and relevant bugs. The binaries span multiple system libraries and hardware contexts, ensuring realistic evaluation scenarios for fuzzers. Finally *G5* is addressed providing a diverse and extensible benchmark set. This prevents overfitting by exposing fuzzers to a variety of binaries, bugs, and conditions rather than letting them specialize on a narrow set of targets.

The resulting FIRMREBUGGER framework is supported by 61 binary targets and 313 software bug oracles within

three sets of compiled, ready-to-fuzz binaries in FIRMBENCH, FIRMBENCHDMA and FIRMBENCHX; with FIRMBENCHDMA and FIRMBENCHX created to capture more complex challenges in fuzzing firmware targets.

Contributions. We make the following contributions:

- We propose a first fuzzing benchmark framework, FIRMREBUGGER, for monolithic firmware fuzzers. Importantly, our approach addresses the problem of leaky oracles, defending against overfitting in benchmarks, and provides ground-truth bug-based metrics.
- Our key idea is to exploit the introspection capabilities of emulators to automatically translate virtual CPU state to bugs states using simple-to-craft expressions of bug descriptions (bug oracles) during replay of fuzzing seeds.
- We implement and open-source FIRMREBUGGER as a *FuzzBench-for-Firmware* type service.
- We curate a benchmark set of 61 diverse binary targets with 313 bug oracles, across three datasets, incorporating fuzzing challenges from our analysis to facilitate fair assessment of current and future advances in firmware fuzzing.

Importantly, we use FIRMREBUGGER to conduct an extensive benchmark study on 9 state-of-the-art firmware fuzzers [**?**, 9, 10, 19, 20, 44–46, 56], across 61 binaries with 187 bugs. Each fuzzer underwent 10 repeated 24-hour fuzzing runs, culminating in a total evaluation effort equivalent to 10 CPU-years.

Paper Organisation. We re-visit approaches to monolithic firmware fuzzing in Section 2. Section 3 investigates *problems* with current empirical performance measures. Our bug-based benchmark framework, FIRMREBUGGER, and its implementation is in Section 4. Importantly, in Section 5 we investigate current *firmware fuzzing challenges facing re-hosting frameworks* to construct a diverse set of targets incorporating the challenges to provide for future advances in firmware fuzzing. Then, in Section 6, we present and analyze the results of our own extensive benchmark study of state-of-the-art fuzzers.

2 Background on Firmware Fuzzing

Monolithic Firmware. Firmware, software for the hardware, is found in embedded systems across industries, ranging from automotive systems, industrial control systems, medical devices, to consumer electronics. Monolithic firmware targets microcontroller units (MCUs). These low-cost MCUs feature fewer resources than desktop processors and use simpler instruction set architectures (ISAs) such as ARM or RISC-V. MCUs incorporate a range of different peripherals such as GPIO, ADC, UART or SPI from a diverse set as illustrated in Figure 1. These peripherals are used for real-time interaction with sensors and actuators, allowing complex products such as CNCs [25] or drones [5] to be built. Monolithic firmware typically lacks the abstraction provided by an operating system, instead managing all critical tasks internally, including hardware initialization, peripheral communication, and event handling. This design tightly couples the firmware with the microcontroller unit (MCU) and its peripherals. Several interfaces are used to interact with peripherals:

❶ **Memory Mapped Input Output (MMIO)** is a mechanism for allocating a dedicated region of the system's memory address space to the registers of peripheral devices. As shown in Figure 1, within the MMIO Peripherals Region, each peripheral device contains multiple memory-mapped registers—control, status, and data registers—that provide direct access and control over the device's state, configuration and data.

❷ **Direct Memory Access (DMA)** facilitates efficient data transfer between peripherals and system memory with minimal CPU involvement. The firmware configures the DMA controller by specifying memory buffers and a transfer size. Once initiated, the DMA controller autonomously moves data between the peripheral and memory. When the transfer is completed, the firmware is notified via an interrupt, after which the CPU can directly process the data in memory.

❸ **Interrupts** are signals generated by hardware to notify the firmware of events that require immediate attention. When an interrupt is triggered, the CPU jumps to the associated interrupt handler and processes the signal, before returning to the prior point in execution.

Firmware Fuzzing. Hardware limitations, resource constraints, and the complexity of embedded system interactions make fuzzing firmware challenging. Recent work has explored approaches to address the complexities of employing fuzzing for testing firmware security. Broadly, these approaches can be split into: i) Hardware-based approaches (on-device fuzzing and hardware-in-the-loop methods) [12, 17, 38, 52]; and ii) emulation-based approaches.

On-device fuzzing, while ideal for execution accuracy, is often impractical due to the limited computational power of embedded devices, which results in poor fuzzing throughput [21]. Similarly, hardware-in-the-loop fuzzing introduces additional inefficiencies, as synchronization between the physical hardware and the emulated environment incurs substantial overhead [44]. Moreover, both approaches suffer from limited scalability, since they require access to specific hardware devices, making it difficult to parallelize or scale experiments across large testbeds. Consequently, researchers have proposed emulation-based approaches to support cross-architecture fuzzing, enabling faster execution speeds and greater control over the fuzzing process.

Fuzzing firmware in an emulation environment poses a number of significant challenges. Successful firmware emulation and fuzzing relies upon: i) accurately handling three critical types of interactions between firmware and hardware:

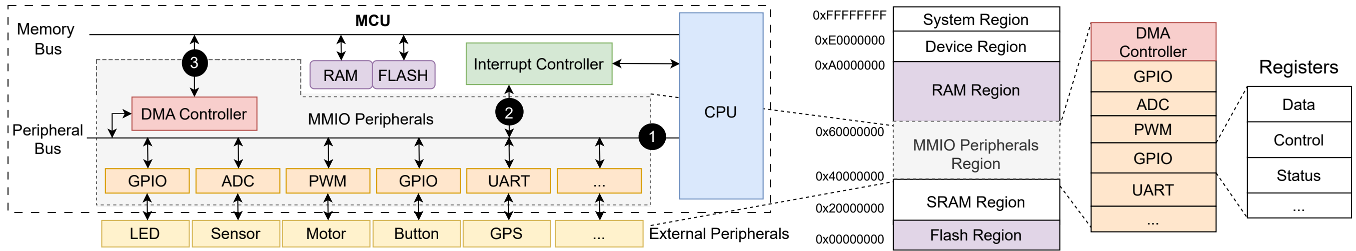

Figure 1: The internal architecture of a microcontroller, illustrating the firmware interactions with peripheral devices.

MMIO, DMA and interrupts; and ii) a fuzzer capable of generating effective fuzz tests to firmware code given the diverse set of peripherals re-hosted code execution can interact with. We assess the effectiveness of solutions to these problems using empirical evaluations, next we investigate challenges faced with empirical performance evaluation of firmware fuzzers.

3 Performance Evaluation Challenges

The performance of fuzzing techniques, including those applied to firmware, is evaluated through *empirical* experiments. Various performance measures, such as code coverage and unique crash counts, are used in the literature to evaluate advances in firmware fuzzing [9, 19, 21, 44, 56]. However, we show that applying such metrics to evaluate firmware fuzzers is fraught with problems. In the following section, we discuss the unique difficulties faced in using established performance measures for monolithic firmware.

3.1 Coverage

Code coverage is a fundamental metric widely employed by firmware fuzzers [10, 19, 21, 44] to assess fuzzing effectiveness. In general, higher code coverage indicates greater exploration of the program's functionality, demonstrating an increasing likelihood of discovering bugs. Thus acting as a proxy for bug discovering ability. Numerous studies have explored the relationship between code coverage and bug discovery [24, 29, 31] supporting this assertion. However, the recent study by Bohme et al. [7] suggested that while there is a notable *correlation* between code coverage and bug discovery, there is no unanimous agreement on whether a fuzzer's superiority is solely determinable by code coverage.

Bug exploitation is a problem that can hinder the collection of accurate coverage information. On conventional targets, mechanisms such as sanitizers (e.g., ASAN [47]), control-flow integrity checks, address space layout randomization, and stronger memory isolation, typically prevent fuzz inputs from causing uncontrolled behavior. Firmware lacks these mechanisms, allowing bugs to result in uncontrolled memory corruption. This can enable the fuzzer to manipulate the program counter by corrupting return addresses or function pointers, which can artificially inflate coverage metrics.

For example, consider the coverage plots in Figure 2. In this experiment, we used FUZZWARE [44] on the Thermostat binary [26] which has one known bug. At first sight, the fuzzer achieves high code coverage (blue) in Figure 2, seemingly indicating excellent performance. However, when examining the execution traces, we observe that a bug exploit allows the fuzzer to reach otherwise unreachable blocks in the code. To show the impact of the bug exploit, we patch the binary to remove the bug and fuzz the patched target. The result, orange plot in Figure 2, shows more than 35% of the original coverage resulted from exploiting the bug. This significant reduction in coverage underscores how a bug exploit can artificially inflate code coverage.

Recent firmware fuzzers have acknowledged the problem. Both MULTIFUZZ [10] and HOEDUR [46] highlight the impact of bug exploitation on fuzzing performance metrics. In MULTIFUZZ [10], 6 of the 23 binaries (27%) tested were impacted by bug exploits. The authors also mention signs of bug exploitation in 2 additional binaries during testing; however, these cases were infrequent and thus excluded from the final results. Similarly, the authors of HOEDUR [46] note that a subset of their tested binaries 12 out of 37 (33%) required patches to mitigate the effects of bug exploitation and preserve the integrity of code coverage as a performance measure.

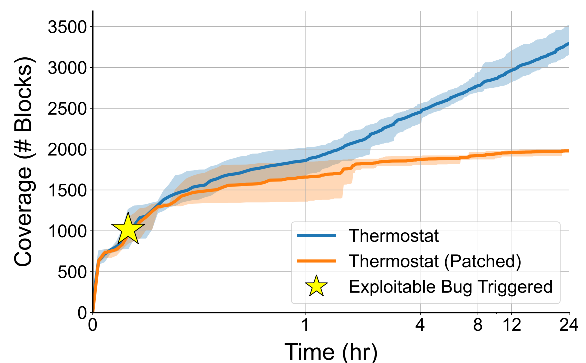

Figure 2: Number of blocks covered by FUZZWARE [44] on the Thermostat (introduced in PRETENDER [26]) vs. the Thermostat (patched) binary (where the exploitable bug is removed) across five 24-hour fuzzing trials. The star denotes the mean time to trigger the exploitable bug.

3.2 Unique Crashes

Counting the number of unique crashes found by each fuzzer is another commonly used metric for evaluating fuzzer effectiveness. In theory, this should be a robust measure, as it directly reflects a fuzzer's ability to discover bugs. However, in practice, relying on unique crashes as a metric is problematic for several reasons. Bugs are inherently complex, and crashes caused by the same underlying bug can manifest through different execution paths, this results in fuzzers saving many duplicate crashes.

To address this, fuzzers employ deduplication heuristics, such as crashing locations, coverage profiles, or stack hashes [23, 49, 53]. Despite this, studies have shown that these heuristics are inaccurate, significantly overestimating (or in some cases, underestimating) the true number of unique crashes [6, 14, 30, 34].

Deduplication Using Common Heuristics. For instance, stack hash heuristics, employed in fuzzers such as Honggfuzz [49], deduplicate crashes based on the call stack. However, with firmware, this heuristic proves less effective due to the nature of interrupts. This is illustrated in Figure 3, which features two crashes caused by the same root cause bug. In Crash 2, the interrupt occurs later resulting in Crash 2 executing an additional function in func_3. Which leads to a disparity in the call stacks and when relying on the call stack to deduplicate crashes this approach would erroneously mark Crash 2 as a unique crash. This effect occurs on a larger scale resulting in significantly inflated unique crash counts. Similarly, methods relying on coverage profiles, such as in AFL [53], for deduplication faces similar challenges and would struggle to deduplicate crashes for the same reasons in firmware targets.

Recent monolithic firmware fuzzers, such as FUZZWARE [44] use the program counter and, link register to deduplicate crashes. Although this heuristic is sufficient in many cases, it can fail in real-world targets. For example, Figure 4 shows two different buffer overflows within the Gateway binary (introduced in P^{2}IM [21]). Despite being two separate buffer overflows, the proximity of the databuffer and pinState buffers introduces a situation where both overflows can corrupt the same variable, uart_handler. **Crash 1** overflows dataBuffer, corrupting uart_handler, while **Crash 2** overflows pinState, corrupting of the same variable. When uart_handler is later accessed in its corrupted state, both crashes will lead to a program crash at the same location. The program counter and link register-based heuristics would incorrectly categorize both cases as the same bug, effectively conflating the two bugs. *This case highlights a flaw with heuristic-based crash deduplication, where distinct bugs can be merged erroneously, leading to missed vulnerabilities.*

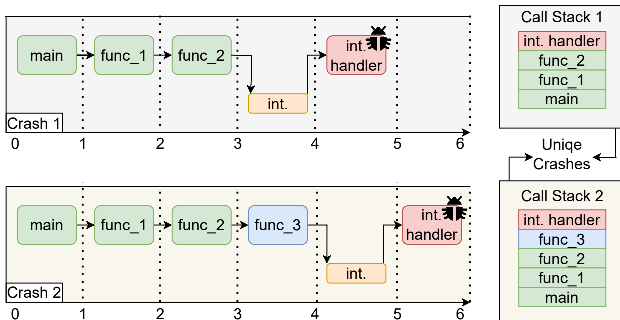

Figure 3: Illustration of two crashes with the same root cause (crash occurs in the interrupt handler). Due to the differences in interrupt timing, Crash 2 has executed an additional function (func 3) and as a result, has a different call stack when compared to Crash 1. Consequently, when using the call stack to differentiate between unique crashes; Crash 1 and Crash 2 are incorrectly distinguished as unique.

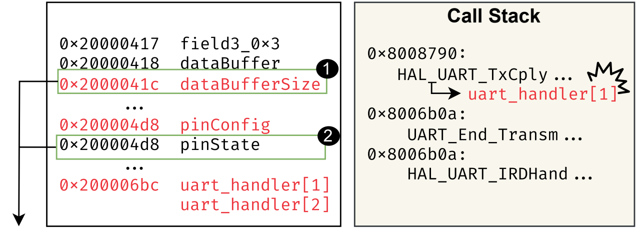

Figure 4: A simplified overview of two bugs from the Gateway binary introduced in P^{2}IM [21]. dataBuffer ① and pinState ② are two different vulnerable buffers defined on the heap. Due to the proximity of the buffers either buffer can overflow and corrupt the same variable—the function pointer uart_handler[1].

3.3 Manual Crash Analysis

Due to the compounding issues we have discussed, many state-of-the-art firmware fuzzers [19, 21, 44, 46, 55] instead manually triage crashes across multiple fuzzers. However, this task is both time-consuming and error-prone.

```
1 //au8Buffer[2] is read from a peripheral
2 for (i=0; i< au8Buffer[2] /2; i++)
3 {
4     //if au8Buffer[2] is large enough
5     //a buffer overflow is triggered
6     au16regs[i] = word(au8Buffer[u8byte],
7                        au8Buffer[u8byte +1]);
8     u8byte += 2;
9 }
```

Listing 1: Code snippet of buffer overflow from the Heat Press binary (introduced in P^{2}IM [21]).

Crash Explosion From a Single Bug. This process can involve analyzing thousands of crashes resulting from a single

bug to determine their uniqueness. Consider the example in Listing 1 from the Heat Press to illustrate a problem increasing the manual triaging task burden. In the binary, a value is read from a peripheral is stored in `au8buffer[2]` and is used as the length for the global `au16regs` buffer. If the length provided exceeds the capacity of `au16regs`, the function can corrupt many different global variables stored after `au16regs` in memory. The corruption of different variables leads to multiple crashes in multiple locations in the code resulting in a massive number of crash reports for a single bug. The potential for bug exploits as described in Section 3.1 further complicates bug triaging and the absence of sanitizers compounds the complexity of analyzing these crashes. This increases the potential for human error and prolongs the triaging process to generate bug-based comparison results to evaluate fuzzers. Such explosion is especially common in firmware, which lacks the memory protections provided by an operating system in traditional applications.

Overwhelming Effort Leads to Missing Bugs. This is exemplified by the RF Door Lock binary (μEMU [55]), where the FUZZWARE [44] authors reported two distinct bugs, FW29 and FW38. In our independent evaluation [22] of crashes resulting from FUZZWARE [44], we discovered an additional bug, FRB01. We believe the miss stemmed from the overwhelming number of crashes ($\approx$ 1600) caused by an easily exploitable bug in the binary. The sheer volume of crashes complicates the triaging process and increases the likelihood of missing bugs. Interestingly, likely for the same reason, other fuzzers that also evaluated this binary—SPLITS [20], HOEDUR [46], EMBER-IO [19], MULTIFUZZ [10], and FUZZWARE-ICICLE [9] overlook bug FRB01.

Key Takeaway: Coverage metrics are susceptible to bug exploitation artifacts. Unique crash counts are susceptible to ambiguities resulting from deduplication heuristics. The current approach of manual crash analysis across multiple fuzzers is labor intensive, error prone, and hinders progress in the field.

4 FIRMREBUGGER

The core idea of our approach to consistently and effectively evaluating fuzzers is to automate the triaging of bugs encountered during fuzzing campaigns on curated benchmark targets using bug oracles. Our proposed benchmarking framework, achieving the aspirational goals in Section 1.2, to automatically curate bug-based metrics is illustrated in Figure 5.

FIRMREBUGGER exists separately from the fuzzing process. We propose replaying both inputs and crashing seeds (Fuzzing Seeds) and automating triaging of bugs. To report on bug statistics FIRMREBUGGER distinguishes and determines the state of each bug encountered with respect to fuzzing inputs. Similar to benchmarks for traditional software such as

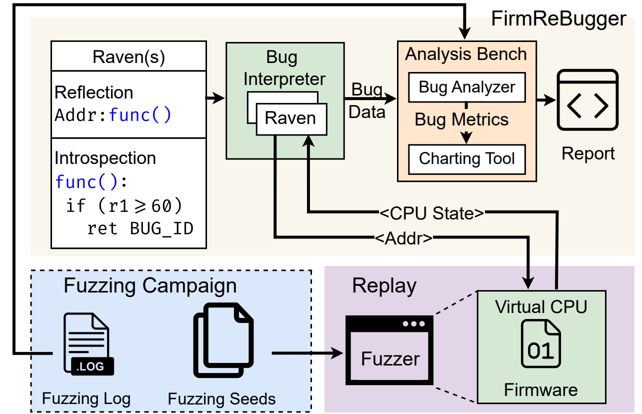

Figure 5: FIRMREBUGGER framework. *Ravens* embed bug oracles as: i) Reflections (virtual CPU states of interest to a bug); and ii) Introspections (C-syntax function descriptions of state introspections to determine bug states). The Bug Interpreter dynamically loads and translates Ravens to bug states, such as Reached & Triggered, and unique bug IDs (Bug Data). The Bug Analyzer in the Analysis Bench processes the bug data along with crash timing data (Fuzzing Log) recorded during a fuzzing campaign into a standardized format, visualized by the Charting Tool to report performance.

FixReverter [54] and Magma [28], our approach also identifies the following states when fuzzing firmware targets:

- **Not Reached.** The input did not lead to executing the code in which the identified bug resides.
- **Reached.** The input executed the lines of code in which the identified bug resides but the inputs did not meet the conditions for the bug to trigger.
- **Triggered.** The input met the conditions to trigger the bug.
- **Detected.** The input triggered the bug and caused a crash.

In particular our approach:

- Isolates the fuzzer (fuzzing logic) from the automated triaging method to ensure the benchmark does not influence the fuzzing process. Effectively, FIRMREBUGGER replays inputs from a fuzzing campaign and utilizes the emulator to gather program state during execution and identify a bug. Importantly, the approach is agnostic to the fuzzer and benchmarking remains independent of the fuzzing logic to avoid the *leaky oracle problem*—*G1*.
- Automates analysis of fuzzing seeds, facilitating the computation of essential metrics such as time to trigger as well as discriminating between bug states (e.g. Reached)—*G2*.
- Curates three dedicated target subsets—FIRMBENCH, FIRMBENCHDMA and FIRMBENCHX—designed to incorporate complex challenges in fuzzing real-world firmware targets to foster continuous growth in fuzzing techniques—*G3* & *G4*.

- Defends against overfitting—where developers optimize for benchmark performance rather than general vulnerability discovery—by providing the means to add new bugs with ease. Incorporating a new bug is as simple as creating an easy-to-craft bug descriptor or a *Raven*—*G5*.

In the following sections, we describe the design of each FIRMREBUGGER component.

4.1 Ravens (Bug Descriptors)

FIRMREBUGGER uses C-syntax to define *Ravens*, to provide ground-truth knowledge of conditions for a bug manifestation during a fuzz input replay.[1] To accommodate the diverse nature of bugs and the complex task of identifying their manifestations and states, we leverage the introspection capabilities of the emulator to inspect the CPU's state at specific points during execution. This process is facilitated by a Raven for each bug in a benchmark binary. A Raven encapsulates the minimum information to uniquely identify a bug's status and is composed of two key components: i) a *Reflection* section; and ii) an *Introspection* section. The schema for constructing a Raven is shown in Listing 2 and we describe a collection of Ravens for a binary.

The reflection section defines locations along the execution path—program counter values represented as `<Address>`—for bug evaluation or data collection. At each reflection point, the associated introspection function—such as `func_1()`—is invoked to analyze relevant state. As shown in Listing 2, these mappings are encoded in the `hook_addresses` array using `context_struct` elements. Each element represents a mapping between a specific program counter (PC) address and the corresponding introspection function to be triggered when execution reaches that address.

At the point of introspection, the invoked function either: immediately evaluates the bug state, or gathers introspection information—such as capturing register values or memory contents at specific program points. This is done through `frb_reg_state` and `frb_mem_read` functions to access both `<Registers>` and `<Memory>` contents, respectively. The collected introspection data is retained and can be referenced by the same or other Ravens later in the replay, enabling the evaluation of more complex or state-dependent bug conditions. The introspection logic invokes either `report_reached` or `report_detected_triggered`; these functions output the *Bug Data*—a bug's state and identifier `<BUG ID>`—. Effectively, a bug is reported as *Reached* if the Raven function is called, and as *Triggered* or *Detected* if the associated `<Logical Expression>` is satisfied. If this logical expression is satisfied by a crashing seed, the bug is recorded as *Detected*. When neither the Raven is called nor the logical expression is satisfied, the bug state is inferred as *Not Reached*.

[1]We feel the songbird, Raven, with a spiritual or godlike status in various cultures and folklore aptly describes a bug oracle role.

```
1  /* Reflection*/
2  context_struct hook_addresses[] = {
3      {<Address>, func_1},
4      ...
5  };
6
7  /* Introspection*/
8  void func_1() {
9      // Report bug Reached state
10     report_reached(<BUG ID>);
11     // Read register of interest
12     <Register> = frb_reg_state(<Register Id>);
13     // Read variable of interest from memory
14     <Memory> = frb_mem_read(<Address>,<Size>);
15     // Evluate bug triggered state
16     if (<Logical Expression>) {
17         //Report bug Triggered/Detected state
18         report_detected_triggered(<BUG ID>);
19     }
20     // Bug not Reached
21 }
```

Listing 2: Schema of a *Raven*. Here `<Logical Expression>` crafted with `<Register>` and `<Memory>` values, encapsulates a bug's triggered status.

4.1.1 Raven Construction and Validation

The workflow for crafting a Raven for a new bug builds upon existing methods for root cause analysis as follows:

1. **Identification.** New bugs manifest as ungrouped crashes (i.e., not matched/identified by an existing Raven).
2. **Root Cause Analysis.** Determine the underlying vulnerability, simultaneously identifying the necessary state conditions for Introspection and execution points for Reflection.
3. **Construction.** Encode the conditions for Introspection and execution points for Reflection into a new Raven with an assigned Bug ID using C-syntax style coding.
4. **Verification.** A Raven's correctness is verified by replaying crashing seeds. A correct Raven must: i) reliably detect the target bug using the intended input (a Raven covers all related crashes); ii) not falsely report distinct bugs (i.e. seeds associated with previously known bugs are not captured by the new Raven); and iii) ensure no crashes remain unlabeled (no ungrouped crashes remain).

Then, we can consider a set of Ravens for a target to be complete if there are no unidentified crashing seeds. Consequently, FIRMREBUGGER aims to achieve no unidentified crashes during the analysis of a target, unless it is a bug for which a Raven remains to be defined.

Describing a bug as a Raven demands a thorough understanding of its root cause through static and dynamic analysis. While root cause analysis is time-consuming (typically ranging from 30 minutes to 4 hours in our experience), it is a stan-

dard prerequisite for responsible bug disclosures. Crucially, the information required for a Raven—the Reflection points and Introspection conditions—is naturally uncovered during this debugging phase. As a result, the subsequent construction of a Raven is generally straightforward, taking approximately 30 minutes.

While automating the root cause analysis process is orthogonal to our work, we have spent considerable effort to curate a comprehensive collection of pre-defined Ravens covering a wide range of binaries grouped into three benchmark sets (FIRMBENCH, FIRMBENCHDMA and FIRMBENCHX), as discussed later in Section 5. Importantly, after root cause analysis of a bug, we have made the process of incorporating a bug with a Raven, simple—we demonstrate the ease with which Ravens can be constructed using three case examples of common bug types:

- Type Confusion
- Stack Buffer Overflow (deferred to our code repository [22])
- Dangling pointer (deferred to our code repository [22])

Type Confusion. This arises when a program erroneously interprets an incorrect type for a region of memory. As a consequence, the program may access fields, invoke functions, or perform operations that are invalid for the underlying data, leading to undefined behavior. Introspection of object types and pointer usage can help identify type confusion bugs.

Raven Example. An illustrative example of a Raven is shown in Listing 3, based on the bug ID `MF01` reported by MULTIFUZZ [10] in the Zephyr SocketCAN binary [55]. In Zephyr's device model, each device is represented by a struct containing a pointer named `driver_api`. This pointer references a table of function pointers that define the operations supported by the device's driver, such as configuration or data transmission. The CAN bus subcommands allow users to specify a target device for command execution. At runtime, the target device is resolved using the `z_impl_device_get_binding` function, which returns a pointer to a generic device struct.

```
1  context_struct hook_addresses[] = {
2      {0x08005e28, BUG_MF04},
3      ...
4  }
5
6  void BUG_MF04() {
7      report_reached("MF04");
8      //canbus fail to verify device type
9      uint32_t read_addr = frb_reg_state[0] + 0x4;
10     if (frb_mem_read(read_addr,4) != 0x0800f7e4){
11         report_detected_triggered("MF04");
12     }
13 }
```

Listing 3: Type Confusion example. A Raven crafted for the bug "MF04" in Zephyr SocketCan binary from μEMU [55].

However, no type verification is performed to ensure that the selected device implements the CAN bus API. As a result, if a non-CAN device is specified (such as a GPIO device), the subcommand will erroneously perform CAN bus operations on an incompatible device struct.

To capture this bug in Raven, the key condition to verify is whether `dev->driver_api` at the point it is used, points to a valid CAN bus API function table. In this context, as shown in Listing 3, whether it matches the address `0x0800f7e4`, which is the base address of the CAN driver's function table. This check should be performed at the point in the CAN bus subcommand function where `dev->driver_api` is about to be invoked (`0x08005e28`). Since `dev->driver_api` is not conveniently held in a register prior to access, its value must be read directly from memory. The introspection function `frb_mem_read` is used to read a specified memory address with the appropriate byte width. In the context of Listing 3, reading from `R0 + 0x4` yields the value of `dev->driver_api`, which can then be compared to the expected CAN API function table address to determine whether the bug is triggered.

We systematically develop 313 accurate Ravens for benchmarking using the method described here by analyzing the bugs in target binaries we discuss in Section 5.

4.2 Bug Interpreter

The *Bug Interpreter* orchestrates the evaluation of each bug during the replay of fuzz inputs—including crashing inputs. By interpreting Ravens with dynamic CPU state observations for every input, the *Bug Interpreter* identifies and records the corresponding bug state and Bug ID. This design decouples the Raven logic from the emulator itself, enabling updates or extensions to Ravens without modifying either the binary under test or the emulator's internal logic.

At the start of each replay session, the *Bug Interpreter* is initialized by dynamically loading Ravens and configuring the emulator with hooks at the reflection points specified by each Raven. During replay, whenever the program counter reaches one of these hooked locations, the *Bug Interpreter* activates the corresponding hook and transfers control to the introspection section. The hook invokes the relevant Raven's logic, potentially leveraging previously gathered introspection data or the current CPU state to assess the bug's status. The resulting *Bug Data*—bug state and corresponding ID—are recorded and forwarded to the *Analysis Bench* to generate bug-based metrics.

Importantly, the Ravens and the Bug Interpreter are agnostic to a target emulator. We provide an API to connect the Bug Interpreter to a given emulator and implement the *Bug Interpreter* using TinyCC [50]. Notably, while many fuzzing approaches exist, the emulators employed are limited to QEMU [4], Unicorn [42] (ported from QEMU) and Icicle [9]. We provide implementations of our Bug Interpreter

API to connect to each of these emulators. Collectively, the attributes supported simplify the adoption of our framework for researchers and developers while Ravens are *portable* across fuzzers.

4.3 Analysis Bench

The *Analysis Bench* assimilates the Bug Data generated by the *Bug Interpreter*, shown in Figure 5, to generate evaluation metrics. As FIRMREBUGGER operates purely as a post-processing step, it can incorporate fuzzing logs from the campaign to extract crash timing information necessary for deriving time-to-bug statistics. The *Bug Analyzer* determines the state of each known bug (not reached, reached, triggered or detected) for every fuzz input and computes the key evaluation metrics such as the *time to reach a bug*, *time to trigger a bug*, and *the number of unique bugs triggered* used in bug-based fuzzing benchmarks [28,54]. Since the program's state cannot be guaranteed after a bug is triggered, only the first triggered bug per input is reported. The rationale for this choice is because a firmware may enter a compromised state after triggering a bug, making subsequent bug reports unreliable (i.e., later reports may depend on the invalid state produced by the first bug). To facilitate further investigation, the bug metrics generated by *Bug Analyzer* flags any crashes that triggered multiple bugs. The analysis generates metrics in a standard, machine-readable format, so the *Charting Tool*—which parses data to generate visualizations and summary statistics—can function as an independent, reusable component for fuzzers or future fuzzing frameworks.

Importantly, once Ravens are added, Ravens and the Analysis Bench are portable across all fuzzers.

5 FirmBench: Benchmark Targets

This section examines the common roadblocks faced by current SoTA monolithic firmware fuzzers. Our findings guide the development of our benchmark target sets.

5.1 Common Roadblocks

Roadblocks are challenges that hinder a fuzzer's progress, reduce throughput, restrict access to key sections of code, or prevent accurate injection of data from the fuzzer into the firmware. Monolithic firmware is riddled with various roadblocks, some commonly encountered in other software such as *magic values* [2] but difficult solve in the firmware domain [20] or those unique to firmware such as interactions with DMA [35,45,56].

We synthesize a set of common monolithic firmware fuzzing roadblocks by analyzing the problems reported and encountered in the nine state of the art fuzzers [10,20,21,35,44–46,55,56]. This involved:

1) Reviewing each of the papers to identify aspects considered out-of-scope and/or unaddressed challenges and limitations.
2) *Crucially*, the practical workarounds found for fuzzing binary targets in associated code repositories, such as source patches, feature selection (e.g., build-flags), and code assumptions.

We discuss the common roadblocks identified, below.

```
1 ...
2     iVar1 = strncmp(argv[1],"settime",7);
3     if ((iVar1 != 0) || (argc != 4)) {
4       iprintf("unknown command...",argv[1]);
5       _rtc_usage();
6       return 1;
7     }
8 // Critical Code Block
```

Listing 4: String Comparison in the Console Binary [21]

Magic Values. Magic values are special constants such as specific strings, numbers or byte sequences that programs use to identify formats, states or to trigger particular behaviors. For fuzzers, these values act as roadblocks, because they enforce strict input checks that must be satisfied before deeper program logic can be reached [2].

This problem is even more pronounced in firmware fuzzing, where inputs are often processed one byte at a time from various peripheral registers through interactions such as interrupts, rather than as contiguous blocks (such as files). This fragmentation makes it more difficult for firmware fuzzers to identify and solve the magic values to make progress [20]. A example from real-world firmware is shown in Listing 4, where a string comparison guards critical code. In prior work [21,44,46,56], source-level patches were applied to bypass such checks in some targets to remove checksums or simplify complex comparisons, confirming that they remain a significant roadblock for state-of-the-art fuzzers.

Complex Peripherals. Fuzzing complex peripherals such as USB or SPI presents unique challenges because both require highly structured, protocol-compliant inputs. Unlike simple data register (DR) accesses, these interfaces demand sequences of interactions that adhere to specific packet formats, command-response patterns, and internal state transitions. Because fuzzers typically generate context-free inputs, the resulting sequences are often infeasible on real hardware. As a result, malformed or incomplete transactions are usually discarded before exercising deeper functionality. In some cases, invalid sequences may trigger crashes that would never occur on real hardware, leading to false positives and reduced fuzzer throughput.

An example interaction with a complex peripheral, USB, where the hardware behavior difficult to model in an emulated environment is shown in Figure 6. The firmware configures

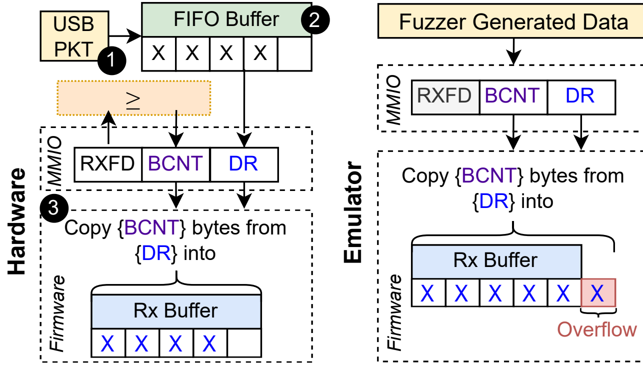

Figure 6: Example USB peripheral interaction in hardware vs. emulation. On hardware, BCNT is always limited by RXFD. In emulation, this check is not enforced, allowing a buffer overflow not possible on the target hardware during the copy.

the RXFD register to set the size of the hardware FIFO buffer and allocates a matching RX buffer in RAM. When a USB packet arrives ❶, it is placed in the FIFO buffer. An interrupt signals ❷ the firmware to read the BCNT (byte count) value from the FIFO Buffer, via the DR register, and copies BCNT-bytes into the RX buffer ❸. The hardware enforces BCNT never exceeds RXFD—and thus the size of the RX buffer—allowing the firmware to assume the value is in bounds. In the emulator, the fuzzer can provide arbitrary or out-of-bounds values for BCNT, ignoring the constraints imposed by the hardware (RXFD $\geq$ BCNT). As a result, the firmware may copy more data than the RX buffer can hold, leading to a buffer overflow that is not possible on real hardware.

Prior work [21, 44, 46, 56] applied source-level patches to simplify or bypass these interactions, such as replacing SPI communication with simplified direct MMIO reads, simplifying BLE and RF chip interactions, or disabling specific interrupts.

DMA. Fuzzing firmware that uses DMA (Direct Memory Access) is challenging because transfers occur between memory and peripherals without CPU involvement. This makes it difficult for fuzzers to determine when DMA transfers should be triggered and where the associated data needs to be injected.

Figure 7 illustrates these challenges. Hardware populates DMA buffers while firmware polls MMIO registers using the `udma_chann...()` function to detect completion. Fuzzers may set MMIO values or trigger interrupts to indicate transfer completion, but without populating the DMA buffer, preventing exploration of DMA-dependent code paths.

DMA is frequently listed as a limitation in prior work, and while some recent studies [45, 56] have proposed techniques to handle DMA interactions, the inherent complexity still poses significant challenges for effective fuzzing.

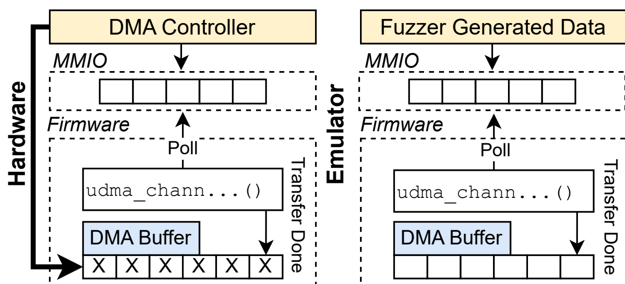

Figure 7: Comparison of DMA in hardware and a fuzzer's emulated environment. The emulator's buffer remains empty.

```
1 __weak void HAL_Delay(__IO uint32_t Delay) {
2   uint32_t tickstart = 0;
3   tickstart = HAL_GetTick();
4   while((HAL_GetTick() - tickstart) < Delay) {
5   }
6 }
```

Listing 5: Example of a delay in the Drone target [21].

Execution Delays and Input Bloating. A major challenge in firmware fuzzing is handling encounters with execution delays and demand for longer inputs. Often, firmware intentionally include various types of delays to synchronize with hardware or wait for events. However, in the context of rehosted emulation-based fuzzing, these delays reduce fuzzing throughput potentially becoming a roadblock.

Another related issue is the growth in fuzz inputs. Input sizes can balloon due to various problems in the fuzzing process, such as inaccurate MMIO emulation and improper handling of interrupts. Larger inputs reduce the fuzzing efficiency as these slow execution and make it more difficult to explore the input space effectively.

A real-world example of the input bloating problem is observed in Listing 5. Each call to the `HAL_Delay` function requires a large number of interrupts to be triggered to advance the tick counter. The impact is two-fold: i) it reduces fuzzing throughput; and ii) if there are MMIO accesses within the triggered interrupt handlers, it inflates the input size needed to make progress.

To mitigate the performance and input bloat effect of execution delays, prior works [20, 21, 44, 55, 56] have adopted various strategies. These include applying source-level patches to shorten or eliminate delay loops, as well as implementing emulation heuristics to detect and fast-forward through delay polling loops.

5.2 Benchmark Target Sets

Considering the fuzzing challenges we discussed, the need to support incremental progress in the field, and and future advances, we curate three benchmarks. To construct a suitable benchmark, we create a set of binaries that not only align with

the design goals ***G3*** and ***G4*** described in Section 1.2, but also satisfy the following criteria:

- **Verifiable.** The binary must contain a confirmed bug, with evidence that the bug is reachable.
- **Non-Trivial.** The bug should be sufficiently complex (>1 hour to be triggered by any fuzzer).
- **Monolithic.** The binaries within the FIRMBENCH set must be monolithic.

The benchmark binary selection draws from a comprehensive review of prior state-of-the-art monolithic firmware fuzzing efforts, further augmented by additional large binaries containing newly identified bugs. We examined all real-world binaries spanning nine published studies to develop the target sets. The collection incorporates an analysis of 50 binaries introduced in SoTA fuzzers, P^{2}IM [21], PRETENDER [26], μEMU [55], HALUCINATOR [11], FUZZWARE [44], DICE [35], SPLITS [20], HOEDUR [46], and MULTIFUZZ [10]. We selected a sample of 30 binaries from the 50 we analyzed in prior works following the criteria above. Further, we expand this set with three new large binary targets containing 15 newly introduced bugs and the discussed roadblocks to curate a total of 33 base binaries. The 33 binaries selected contain 187 unique software bugs we employed to construct the three benchmark target sets detailed below. Importantly, by carefully and systematically analyzing each bug, we developed accurate Ravens for every bug in each benchmark set using the methods in Section 4.

- FIRMBENCH: To establish a standardized baseline for fuzzing comparisons, without the impediments from the roadblocks, we include 28 binaries selected from prior work. Additionally, we incorporate the three large binaries introduced in this work; consistent with prior binaries, we apply source-level patching to remove challenging roadblocks. The curated dataset comprises of 166 bugs, each with a corresponding Raven, spread across 31 binary targets.
- FIRMBENCHDMA: Managing DMA injected data remains a key challenge, this set of binaries aim to support the development of fuzzers addressing the problem. The set is built by filtering the 33 selected binaries to include only those containing DMA interactions and reverting patches that replaced DMA interactions with equivalent MMIO operations in those binaries included from prior work. The curated set comprises 31 bugs, each with a corresponding Raven, spread across 8 binary targets.
- FIRMBENCHX. In our benchmarking efforts, we recognize that the use of fuzzing-specific binaries—where certain challenges are patched or removed—can unfairly disadvantage fuzzers designed to tackle these very problems. To address this, we introduce a challenging benchmark set, FIRMBENCHX, in which binaries are left untouched with respect to the roadblocks we discussed. We make patches/modifications only when absolutely necessary to enable the binary to execute in an emulator. These patches address emulator capability limitations (unrelated to the roadblocks), such as substituting any hardware-assisted implementations of functions like `memcpy` with their software equivalents. This policy of minimal intervention preserves the inherent complexity of the original binaries. The binaries curated offer a challenging, realistic and rigorous benchmark for future advances in fuzzers. The curated set comprises 109 bugs, each with a corresponding Raven, spread across 22 binary targets.

Notably, we retain false positive bugs. A false positive occurs when a fuzzer reports a crash that arises from an emulated state that cannot occur on real hardware. Although false positive bugs are undesirable, they can demonstrate a fuzzer's ability to reach deep execution paths and uncover latent vulnerabilities [44]. We clearly identify false positives, discussing them in detail in Appendix C.

5.2.1 Analysis of the Benchmark Binary Targets

FIRMBENCH encompasses 187 unique bugs and 313 Ravens across 34 distinct CWEs, as summarized in Figure 8(a). To analyze the benchmark, for each bug, we assign a single

Figure 8: (a) Distribution of CWEs across unique bugs, excluding FPs and (b) roadblocks across the sample of 33 binaries.

CWE following the guidelines provided by MITRE [37]. In cases where bugs have associated CVEs, we adopt the CWE specified in the corresponding CVE record. The selected 33 binaries along with the roadblocks reflected in these binaries, and any associate bug description CWEs are summarized in Table A1. We detail the modifications made to the selected binaries from prior work to construct the 61 binaries (FIRMBENCH: 31, FIRMBENCHDMA: 8, FIRMBENCHX: 22) used in the benchmark sets in our code repository [22].

As illustrated in Figure 8(b), the 187 unique bugs in 33 firmware targets exhibit diverse bug categories (34 different CWEs) and with significant representations of roadblocks: 28 (85%) implement magic-value checks, 24 (73%) interact with complex peripherals, 10 (30%) leverage DMA, and 32 (97%) contain delay or bloat code. Collectively, these targets cover 14 distinct microcontroller and 10 unique system libraries as detailed in Table A1. Notably, the analysis reflects the current state of the benchmark but FIRMREBUGGER supports the community to expand the benchmark sets as new bugs are discovered.

6 Experimental Evaluations

We evaluated the performance of state-of-the-art fuzzers to establish the versatility of our benchmark suite and understand the effectiveness of techniques explored in recent studies.

Fuzzers. We selected nine state-of-the-art firmware fuzzers for evaluation: FUZZWARE [44], EMBER-IO [19], FUZZWARE-ICICLE [9], SEMU [56], DICE [35], GDMA [45], SPLITS [20], HOEDUR [46], and MULTIFUZZ [10]. Notably, not all targets are executed on SEMU since it requires manual, platform-specific setup for new targets. Additionally, due to unimplemented instructions within the version of Unicorn [42] used by FUZZWARE and SPLITS, we exclude the Oresat-Control binary from the evaluation of these fuzzers. Then, for FIRMBENCHDMA, we employ the recent fully-automated fuzzers [35,45] focusing on extending the existing firmware fuzzers to support DMA using information gathered at runtime[2]. We defer extended details of the evaluation rationale and settings to Appendix D.

Benchmark Metrics. We report: i) *bug count*—the number of unique bugs triggered at least once across 10 independent trials; ii) *time-to-bug* and survivability functions following the method in Magma [28]; and iii) we evaluate each fuzzer's *consistency* to measure the reliability with which bugs are discovered across repeated trials where consistency is defined as:

$$\text{Consistency}(f) = \frac{1}{|B|}\sum_{b\in B}\left(\frac{c_{f,b}}{T}\right),$$

where, f is a particular fuzzer, B is the set of all bugs, $|B|$ is the total number of bugs, $c_{f,b}$ is the number of trials in which fuzzer f triggered bug b, and T is the total number of trials.

Evaluation Regime. We execute each fuzzer for 24-hours and use 10 trials on each binary target. This is a significant effort, totaling 10 CPU-years of computation time.

7 Results and Discussions

In this section, we discuss the versatility and utility of FIRMREBUGGER as a benchmark, and investigate the effectiveness of various existing fuzzers based on our results.

Figure 9 and 10 summarize the bug finding ability of fuzzers in upset plots; the vertical bars represent the num-

[2]Although a concurrent study to GDMA investigating fully automated methods of data injection to DMA, DYMA-FUZZ [18], was recently published, at the time of benchmarking, the method was unavailable. We defer its integration to the benchmark to our code repository.

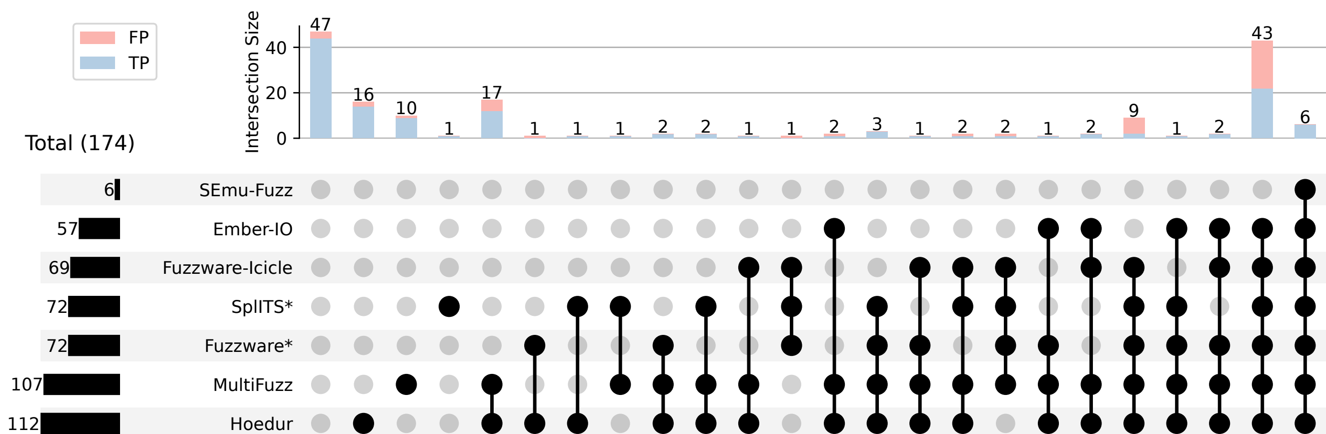

Figure 9: Distribution of true and false positive bugs across fuzzers over 10 trials (24h) on the FIRMBENCH benchmark set. Note: fuzzers marked with an asterisk (*) were evaluated on a subset of binaries–we detail our rationale in Appendix D.

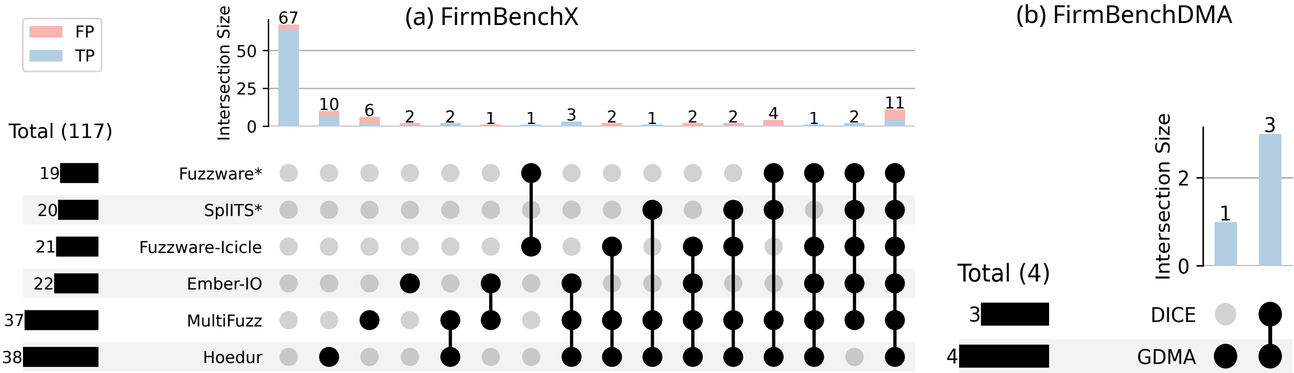

Figure 10: Distribution of true and false positive bugs across fuzzers over 10 trials (24h) on the (a) FIRMBENCHX and (b) FIRMBENCHDMA benchmarks. Fuzzers marked with an asterisk (*) were evaluated on a subset of binaries—see Appendix D.

ber of triggered bugs in each intersection, the horizontal bars show the total number of bugs triggered per fuzzer, and the connected dots denote the fuzzers involved in each intersection. Figure 11 illustrates consistency, Figure 12 presents a selection of survivability plots with Table 1 showing the generated time-to-bug metrics for FIRMBENCHDMA (full results are available on our GitHub repo [22]).

7.1 Observations

Overall Analysis. Across our three benchmark sets, comprising a total of 295 bugs, the evaluated fuzzers triggered 181 bugs overall: 127 of 174 in FIRMBENCH, 50 of 117 in FIRMBENCHX, and 4 of 4 in FIRMBENCHDMA. As shown in Figure 9 and Figure 10 (a), MULTIFUZZ and HOEDUR are the best performing fuzzers, triggering 144 and 150 bugs, respectively. In contrast, single-streamed fuzzers performed notably worse, with similar results to each other—except for SEMU. SPLITS performed marginally better than the others by uncovering bugs guarded by solving magic value comparisons.

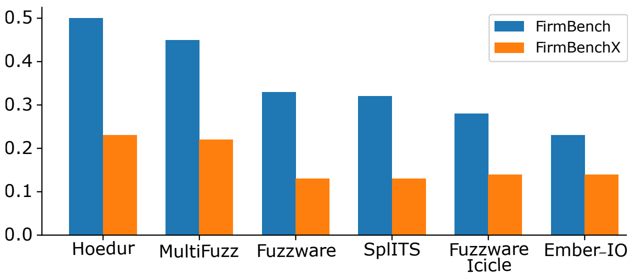

Figure 11: Consistency score comparison. Higher scores generally indicates greater bug-finding reliability. But it should be interpreted alongside the number of bugs triggered as consistency alone does not capture the breadth of bug coverage.

Figure 11 summarizes the consistency of each fuzzer at triggering bugs. Comparing the two leading fuzzers, overall, HOEDUR demonstrates higher consistency than MULTIFUZZ. We hypothesize that HOEDUR's improved consistency may be due to its use of probability matching (Thompson Sampling [8]) to adaptively select data streams for mutation based on their observed success rates in increasing coverage [46]. This approach aims to avoid being misguided by frequent but uninformative register accesses.

Our evaluation on FIRMBENCHX in Figure 10 (a) shows a significant drop in bugs triggered—from 73% in FIRMBENCH to 43% in FIRMBENCHX. This stark decrease highlight a research gap for future developments to tackle the challenges and roadblocks we discussed in Section 5.1.

Key Takeaway: 73% of the bugs within FIRMBENCH were triggered by at least one fuzzer within 24 hours, while only 43% of the FIRMBENCHX bugs were triggered. As expected, recent fuzzers employing multi-stream input representations [10, 46] yielded the best performance while stream-specific mutations may improve consistency.

Magic Values. As shown in Figure 12, for the Console binary and bug I01, only HOEDUR, MULTIFUZZ, SPLITS, and FUZZWARE-ICICLE are able to trigger the bug, with FUZZWARE-ICICLE doing so only two times, while the others achieve full coverage. FUZZWARE-ICICLE uses Compare Coverage (CompareCov) feedback to guide mutations near comparison instructions, which enables it to occasionally solve guarded magic values checks, but with less consistency than other approaches. In contrast, HOEDUR combines a dictionary-based mutation strategy but with its multi-stream approach, it is able to efficiently generate and match magic value sequences more often. Notably, both SPLITS and MULTIFUZZ utilize input-to-state feedback mechanisms tailored for firmware, enabling them to closely correlate input mu-

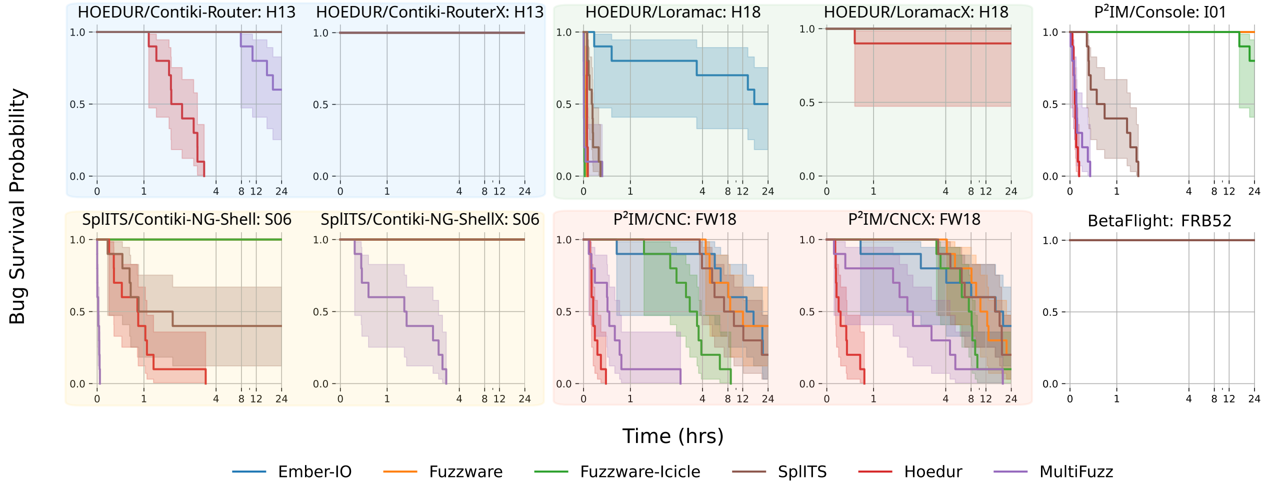

Figure 12: Survival functions for selected bugs. The shaded areas indicate 95% confidence intervals. Colors group pairs of related binaries (e.g., original and 'X' variant) for visual comparison (statistics and plots generated by our Analysis Bench—see Fig. 5).

tations with program state changes and systematically overcome magic value conditions. This is a likely source of bugs uniquely identified by each of these fuzzers (Figure 9).

Key Takeaway: Magic values are used in firmware and can guard bugs. Input-to-state mechanisms [10, 20] or dictionaries (combined with multi-stream inputs) [46] prove effective at consistently overcoming these roadblocks.

Complex Peripherals. As shown in Figure 12, the Loramac and LoramacX binaries with bug H18 differ significantly in their SPI handling. In Loramac, the binary is patched with a direct SPI read in the `transceive` function, eliminating explicit interaction with the peripheral, whereas in LoramacX this patch is reverted bring the interaction back. This makes fuzzing substantially more challenging for two reasons: it introduces a dependency on realistic peripheral behavior—something generic fuzzers often struggle to emulate—and it requires the fuzzer to supply specific, meaningful input via the SPI interface to trigger the bug. This not only expands the input space but also increases the complexity of exercising the bug condition. As a result, the challenge binary (Loramac) is significantly harder for fuzzers: as evidenced by the survivability graph in Figure 12, only HOEDUR was able to trigger the bug—and only once out of ten trials—whereas previously, the majority of fuzzers could do so consistently.

Key Takeaway: Complex peripheral interactions prevent many fuzzers from reaching bugs in a timely manner; impacting even the most recent multi-stream methods.

DMA. Consider bug H13, shown in Figure 12, the challenge version (Contiki-RouterX) reintroduces a patch restoring DMA handling in the RF packet reception path. In the original binary, both HOEDUR and MULTIFUZZ could reliably trigger the bug. However, the DMA patch fundamentally alters how input data is processed: received radio packets are now transferred directly into memory via DMA, bypassing the traditional, byte-wise CPU-driven reads. Consequently, the fuzzers are blind to the data path that reaches the vulnerable code, preventing these fuzzers from triggering the bug, as shown in Figure 12.

For the DMA-focused fuzzers DICE and GDMA, our findings—summarized in Figure 10 (b)—indicate that GDMA [45] outperforms DICE [35] on our FIRMBENCHDMA targets. Table 1 presents the median reached and trigger times across bugs for both fuzzers. GDMA [45] consistently achieves faster times, particularly on the Modbus target (e.g., DI2 and DI3), where it reaches and triggers the bugs in under 10 minutes compared to DICE's [35] 55-79 minutes. While trigger rates are comparable on some bugs, the reduced time-to-bug demonstrates that GDMA [45] exercises DMA-related code paths more efficiently.

Binary	Bug ID	GDMA [45]			DICE [35]		
		Med. R	Med. T	Hit	Med. R	Med. T	Hit
MIDI-Synthesizer	DI4	00:13	00:13	100%	**00:08**	**00:08**	100%
	DI5	**00:15**	**00:15**	100%	00:16	00:16	100%
Modbus	DI2	**00:09**	**00:09**	100%	00:55	01:10	90%
	DI3	**00:09**	–	40%	01:19	–	0%

Table 1: Median bug survival times—both **R**eached and **T**riggered—over a 24-hour period across 10 trials for the FIRMBENCHDMA set, reported in HH:MM. "Hit" denotes the percentage of trials in which the bug was successfully triggered. The best-performing times are in bold. Table and statistics were generated using the Analysis Bench (see Fig. 5) while other time-to-bug results are deferred to [22].

Key Takeaway: Fuzz data injection into DMA remain challenging—with only limited developments [18,35,45] pursuing fully automated methods—because DMA controller implementations vary significantly by manufacturer or product lines. But, DMA inputs can guard critical code paths and are prevalent in firmware [45] (10 binaries in our FIRMBENCHDMA set include DMA as a roadblock).

Execution Delays and Input Bloating. The roadblock in the CNC and CNCX binaries with bug FW18 is through a delay function that is reinstated in the challenge version. This delay function uses a busy-wait loop, forcing fuzzers to wait for the delay to elapse. This additional timing constraint causes all fuzzers to take longer to trigger the bug in the challenge binary as seen in Figure 12.

Meanwhile, in the Contiki-NG-Shell and Contiki-NG-ShellX binaries with bug S06 (see Figure 12, Contiki-NG-ShellX) reintroduces the `fade` function, that executes thousands of loop iterations and MMIO accesses. This function requires the fuzzer to provide sufficiently long inputs in order to progress through the entire loop and any triggered interrupt handlers before reaching the buggy code path. As a result, only fuzzers with explicit length-extension strategies—as seen with MULTIFUZZ—are able to consistently trigger the bug in the challenge binary. Other fuzzers frequently stall within the loop.

The BetaFlight binary with bug FRB52, as illustrated in Figure 12, incorporates all of the identified roadblocks, making it exceptionally challenging to uncover the bug. Consequently, all existing fuzzer are unable to trigger the bug. Discovery of the bug requires manual analysis, patching, and providing far more than 24 hours of fuzzing time.

Key Takeaway: The use of functions that, for example, execute excessive loop iterations—particularly for accessing MMIO—hinder a fuzzer from making progress to discover bugs. These functions can reduce test throughput, or lead to bloated inputs making mutations less effective.

False Postive Bugs. Although false positive bugs are undesirable, they can demonstrate a fuzzer's ability to reach deep execution paths [44]. Therefore, we retain false positive bugs in FIRMBENCH, clearly marking them as such. Notably, some false positives are specific to certain fuzzers, which should be considered when interpreting results. We discuss false positives further in Appendix C.

7.2 Extending With Community Contributions

Importantly, we envision a continuing expansion of FIRMBENCH through submission of Ravens, preventing overfitting and allowing for continued effective evaluation as fuzzers develop. Ravens submitted with a thorough root cause analysis of new bugs (similar to bug disclosures), where the community validates the Raven accurately encapsulates the bug's semantics, can be merged into FIRMBENCH. In addition to manual reviews, accurate encapsulation can be validated by ensuring the Raven correctness as described in Section 4.1.

We acknowledge the potential for community extensions to bias the benchmark targets or bugs in FIRMBENCH towards, for instance, a technique for a particular roadblock. While we have used some foresight and curated a DMA specific benchmark set for such an eventuality, the evolution of the field and the need to evaluate techniques may drive the creation of such subsets with bugs guarded by roadblocks. We envision the formation of a balanced binary set, that includes a balanced set of bugs from various categories and roadblocks as the field evolves. This would allow evaluation both at a wider level, and for techniques addressing specific bug types or roadblocks. The framework we developed is flexible to support these needs.

7.3 Potential FIRMBENCH Dataset Bias

The evaluated binaries are primarily sourced from prior works. This can potentially bias results towards these fuzzers. To minimize bias towards any one fuzzer, we curate our bug benchmark from bug reports across many different fuzzing works. As discussed in Section 5, we considered 10 SoTA works and add 3 new binary targets with 15 unique bugs whilst curating three target sets. Further, bug diversity, as discussed in Section 5.2.1, covering 34 unique CWEs prevents fuzzers targeting any one specific bug type from being unfairly advantaged and acts to mitigate bias. Importantly, FIRMBENCH is built to be extensible, facilitating the transparent introduction of new bugs over time to further mitigate biases and overfitting as discussed in Section 7.2.

Additionally, FIRMREBUGGER consistently provides time-to-bug information for comparing fuzzers, a metric often not included in fuzzer evaluation [10, 19–21, 35, 44, 55, 56]. Time-to-bug measure allows works to demonstrate improved speed and consistency in triggering known bugs, beyond that observed for the fuzzer that originally discovered the bug.

7.4 Triggered vs Detected

We report bug count metrics based on the Triggered status because it can capture how well a fuzzer can explore code paths and program state of the space for bugs for a target. Whilst the detected status could be used, it can lead to under reporting bug finding capability as not all bugs triggered may result in a crash within a fuzzing campaign. It is important to acknowledge and appreciate that Triggered status based metrics may be influenced by differences in the emulators crash detection measures, or a fuzzer's seed-saving strategy, since FIRMREBUGGER replays saved seeds and crashes. To avoid the possible impacts from differences, we support a LiveMode

operation as we discuss in Section 7.5. Alternatively users of the benchmark can consider results based on both Detected and Triggered status when reporting bug counts.

7.5 FirmReBugger *Live-Mode* Debugging

Importantly, to enable interactive investigation of bugs, we implement a *Live-Mode* of operation that can be employed during fuzzing by exploiting our Ravens and the provided API for emulators. We allow Ravens to be active or inactive. Active Ravens function as safeguards to identify if a triggering condition for a bug during execution is satisfied, and immediately crash. This allows a fuzzer to search for new vulnerabilities without the impact of previously known bugs, such as from bug exploits, making FIRMREBUGGER a flexible tool for targeted debugging and analysis.

7.6 Related Work

The need for methods to fairly assess fuzzers has led to the creation of benchmarking frameworks, *outside* of the firmware domain, notably with Klees et al. [30], LAVA [16], Magma [28], UNIFUZZ [32], FuzzBench [36], FixReverter [54], and FuzzProBench [39]. These suites aim to provide curated collections of binaries with realistic bugs to directly evaluate *bug-finding* capabilities through bug counts and time-to-bug metrics. Prior work to design benchmarks considered dynamic injection of synthetic bugs, as in LAVA [16], manually reverting bug patches as in Magma [28] and the DARPA Cyber Grand Challenge [13], and automatic patch reversion, as in FixReverter [54]. Notably, representativeness of synthetic bug injections remains debated [16, 28], further FixReverter depends on static analysis to determine bug reachability—a difficult task in firmware characterized by complex hardware interactions. Ours, similar to Magma, considers real-world targets and bugs but address the leaky oracle problem [28] by proposing a means for automatic triaging during replay of fuzz inputs without modifying the targets.

8 Conclusion

FIRMREBUGGER is a bug-based benchmark designed to address the distinct challenges of monolithic firmware fuzzing. With carefully constructed real-world binaries and a standardized and automated evaluation workflow, we demonstrate and offer the community a reliable method for rigorous, reproducible assessment of monolithic firmware fuzzing techniques. Looking ahead, we envision our benchmark evolving alongside the field, accommodating new insights and techniques as they emerge. Ultimately, we hope FIRMREBUGGER will help speed-up developments and streamline evaluation.

Acknowledgements

The work was supported by the Cyber Security Research Centre Limited whose activities were partially funded by the Australian Government's Cooperative Research Centres Programme.

A Ethical Considerations

This work presents a benchmark for evaluating firmware fuzzing tools. Our intent is to support the security community in developing and evaluating effective fuzzing techniques, thereby improving the security of embedded systems to improve the safety and security of end-users of systems and devices with embedded systems.

In considering the datasets for the benchmark, we decided to employ firmware images either open-source or publicly released, and no proprietary or personal data is included. We examined past state-of-the-art studies with public scrutiny of their work to source these firmware images. Consequently, the sourced software bugs were known and disclosed to project owners and maintainers.

We acknowledge that the benchmark and tools developed in this work have dual-use potential: while they are designed to support security researchers and improve the security of embedded systems, the tool developed from or tested with FIRMREBUGGER to improve their effectiveness could also be misused to identify vulnerabilities for malicious purposes by adversarial actors. These risks are partially reduced by our decision to employ a dataset of open-source or publicly available firmware and software bugs disclosed responsibly for active projects with support from maintainers.

Further, we also acknowledge that fuzzing the binaries used in our benchmark may lead to the discovery of previously unknown vulnerabilities. When such cases arise, these should be disclosed following responsible disclosure practices; reporting these findings directly to the relevant authors or maintainers prior to any public release or inclusion in the benchmark and providing sufficient time for maintainers to address the bug and release patches.

Ultimately, we believe, along with the precautions and commitment to following responsible disclosure practices for software bugs, the benefits of advancing defensive research through open science and strengthening system security outweigh the risks.

B Open Science

To promote transparency and foster further advancements in firmware fuzzing research, we have open-sourced all code and materials associated with this work. These artifacts are publicly available on https://github.com/FirmReBugger/FirmReBugger.

References

[1] Arduino. Arduino: Open-source electronics platform. https://www.arduino.cc/, 2024.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Network and Distributed System Security Symposium (NDSS)*, volume 19, pages 1–15, 2019.

[3] Richard Barry and FreeRTOS Community. Freertos: Real-time operating system for microcontrollers. https://www.freertos.org/, 2024.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46, 2005.

[5] betaflight. Link to betaflight binary. https://github.com/betaflight/betaflight.

[6] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. Aurora: Statistical crash analysis for automated root cause explanation. In *USENIX Security Symposium (USENIX Security)*, pages 235–252, 2020.

[7] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *International Conference on Software Engineering (ICSE)*, pages 1621–1633, 2022.

[8] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. *Advances in Neural Information Processing Systems (NeurIPS)*, 24, 2011.

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

[10] Michael Chesser, Surya Nepal, and Damith C. Ranasinghe. MultiFuzz: A Multi-Stream fuzzer for testing monolithic firmware. In *USENIX Security Symposium (USENIX Security)*, 2024.

[11] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *USENIX Security Symposium (USENIX Security)*, 2020.

[12] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception:System-Wide security testing of Real-World embedded systems software. In *USENIX Security Symposium (USENIX Security)*, pages 309–326, 2018.

[13] DARPA. Darpa cyber grand challenge (cgc) binaries. https://github.com/CyberGrandChallenge/.

[14] Sanjeev Das, Kedrian James, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. A flexible framework for expediting bug finding by leveraging past (mis-) behavior to discover new bugs. In *Annual Computer Security Applications Conference (ACSAC)*, pages 345–359, 2020.

[15] Contiki-NG Developers. Contiki-ng: The os for next generation iot devices. https://contiki-ng.org/, 2024.

[16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[17] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing embedded systems using debug interfaces. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[18] Guy Farrelly, Michael Chesser, Seyit Camtepe, and Damith C. Ranasinghe. DyMA-Fuzz: Dynamic direct memory access abstraction for re-hosted monolithic firmware fuzzing. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026.

[19] Guy Farrelly, Michael Chesser, and Damith C. Ranasinghe. Ember-IO: Effective firmware fuzzing with model-free memory mapped IO. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*, 2023.

[20] Guy Farrelly, Paul Quirk, Salil S. Kanhere, Seyit Camtepe, and Damith C. Ranasinghe. Splits: Split input-to-state mapping for effective firmware fuzzing. In *European Symposium on Research in Computer Security (ESORICS)*, 2023.

[21] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *USENIX Security Symposium (USENIX Security)*, pages 1237–1254, 2020.

[22] FirmReBugger. Firmrebugger: A benchmark framework for monolithic firmware fuzzers. https://github.com/FirmReBugger/FirmReBugger.

[23] google. Clusterfuzz. https://google.github.io/clusterfuzz/.

[24] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *International conference on software engineering (ICSE)*, pages 72–82, 2014.

[25] GRBL. Grbl. https://github.com/grbl/grbl.

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

[27] Steve Hanna, Rolf Rolles, Andrés Molina-Markham, Pongsin Poosankam, Kevin Fu, Ari Juels, and William Powell. Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In *USENIX Workshop on Health Security and Privacy (HealthSec)*, San Francisco, CA, USA, 2011.

[28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[29] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International conference on software engineering (ICSE)*, pages 435–445, 2014.

[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2018.

[31] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE international conference on software analysis, evolution, and reengineering (SANER)*, pages 560–564, 2015.

[32] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security Symposium (USENIX Security)*, pages 2777–2794, 2021.

[33] Arm Limited. Mbed os: The open source operating system for iot. https://os.mbed.com/, 2024.

[34] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

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

[36] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1393–1403, 2021.

[37] MITRE. Cve → cwe "root cause mapping" quick tips. https://cwe.mitre.org/documents/cwe_usage/quick_tips.html.

[38] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research, colocated with NDSS Symposium (BAR)*, volume 18, pages 1–11, 2018.

[39] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[40] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152, 2015.

[41] The Zephyr Project. Zephyr real-time operating system (rtos). https://zephyrproject.org/, 2024.

[42] NGUYEN Anh Quynh and DANG Hoang Vu. Unicorn: Next generation cpu emulator framework. *BlackHat USA*, 476, 2015.

[43] RIOT Community. Riot - the friendly operating system for the internet of things - supported cpus. https://www.riot-os.org/cpus.html, 2025.

[44] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *USENIX Security Symposium (USENIX Security)*, pages 1239–1256, 2022.

[45] Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, Simon Woerner, Daniel Klischies, Felix Buchmann, Nils Ole Tippenhauer, Thorsten Holz, Marius Muench, and Reviewing Model. Gdma: Fully automated dma rehosting via iterative type overlays. In *USENIX Security Symposium (USENIX Security)*, 2025.

[46] Tobias Scharnowski, Simon Woerner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. Hoedur: Embedded firmware fuzzing using multi-stream inputs. In *USENIX Security Symposium (USENIX Security)*, 2023.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Security Symposium (USENIX Security)*, pages 309–318, 2012.

[48] Giovanni Di Sirio and ChibiOS Developers. Chibios: Real-time embedded operating system. https://www.chibios.org/, 2024.

[49] Robert Swiecki. honggfuzz, 2016.

[50] TinyCC. Tinycc: A small and lightweight c compiler. https://repo.or.cz/w/tinycc.git.

[51] uTasker Project. utasker: Embedded operating system and tcp/ip stack. https://www.utasker.com/, 2024.

[52] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS)*, volume 14, pages 1–16, 2014.

[53] Michal Zalewski. American fuzzy lop (afl), 2015.

[54] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *USENIX Security Symposium (USENIX Security)*, pages 3699–3715, 2022.

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

[56] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3269–3283, 2022.

C False Positive Bugs

False positives are common when emulating embedded systems due to the close interaction between firmware and hardware and the inherent inaccuracy of emulation. A false positive occurs when a fuzzer reports a crash that arises from an emulated state that cannot occur on real hardware. A common source of these erroneous states come from interrupt injection. Emulators often fire interrupts pseudo-randomly, producing states that are impossible on actual devices.

False positives impact fuzzing campaigns in several ways. Crashing seeds influence the fuzzer's feedback and seed scheduling, while the increased number of crashes increases triage effort. Although false positive bugs are undesirable, they still demonstrate a fuzzer's ability to reach deep execution paths and find bugs. Furthermore, as highlighted by FUZZWARE [44], these false positives often uncover hidden assumptions at the hardware/firmware boundary. Assumptions that may not always hold across different hardware contexts. This makes such findings valuable, as they can reveal potential vulnerabilities if the firmware is deployed on different hardware. Therefore, we retain false positives in our results, clearly marking them as such. Notably, some false positives are specific to certain fuzzers, which should be considered when interpreting the results.

D Benchmark Evaluation and Setup

FIRMBENCH. For our baseline evaluations, we select seven recent state-of-the-art firmware fuzzers: EMBER-IO [19], FUZZWARE [44], FUZZWARE-ICICLE [9], SEMU [56], SPLITS [20], HOEDUR [46], and MULTIFUZZ [10]. Where applicable, we use FUZZWARE's [44] configuration files as the definition for the target's memory map. For EMBER-IO [19] and SEMU [56] we generate an equivalent configuration in a supported format. However, as SEMU [56] requires manual, platform-specific set up for new targets, its evaluation is limited to the subset of binaries with support already available for target of boards: F103, F429, K64 and SAM3X. Additionally, due to some instructions not being implemented within the version of Unicorn [42] used by FUZZWARE and SPLITS, we exclude the Oresat-Control—see Table A1—binary from the evaluation of these fuzzers.

FIRMBENCHDMA. Two recent fully-automated works [35, 45] focus on the extending existing firmware fuzzers to support DMA using information gathered at runtime. To allow for comparison of these works, we include two bug-containing DMA binaries compatible with both GDMA and DICE.

FIRMBENCHX. For evaluations with FIRMBENCHX, we consider the fuzzers used with FIRMBENCH, except SEMU [56] due to SEMU's dependence on additional manually configured components, and the limited exploration established in FIRMBENCH, we exclude SEMU from the FIRMBENCHX tests.

E Benchmark Bug Descriptions

Table A1: Summary of fuzzing roadblocks identified in each binary in the FIRMBENCH benchmark. Symbols indicate subset inclusion: † FIRMBENCHX, ‡ FIRMBENCHDMA, and ∗ present across all benchmark subsets. We use the presence of calls to `strcmp`, or `strncmp` as a proxy for the presence of magic values. Complex peripherals are determined based on access to SPI or USB interfaces. DMA is ticked for any binaries that perform transactions with the DMA controller. Delay & Bloat is ticked if functions such as `fade` or `delay` are called that bloat inputs or intentionally delay further execution by executing loops without any other purpose

REF	Binary	Blocks	OS/Sys lib.	Magic Value	Cplx. Peripherals	DMA	Delay & Bloat	BugID (CWE)
P2IM [21]	CNC †	3615	Bare metal	✗	✓	✗	✓	**FP_E02**, **FW11**(CWE-121), **FW18**(CWE-1286)
	Console †	2225	RIOT [43]	✓	✗	✗	✓	**I01**(CWE-1284)
	Gateway †	4922	Arduino [1]	✓	✓	✗	✓	**E01**(CWE-252), **FP_FRB01**, **FP_FRB02**, **FP_FRB03**, **FP_FRB04**, **FP_FRB10**, **FP_FW21**, **FP_FW22**, **FW12**(CWE-787), **FW23**(CWE-825), **MF01**(CWE-1284)
	PLC †	2304	Arduino [1]	✗	✓	✗	✓	**FP_FW25**, **FW14**(CWE-1285), **FW15**(CWE-1285), **FW16**(CWE-1285), **FW17**(CWE-1285)
	Soldering_Iron	3657	FreeRTOS [3]	✗	✓	✗	✓	**FP_FRB05**, **FP_I02**, **FP_MF02**, **FW19**(CWE-825), **H01**(CWE-825)
PRETENDER [26]	RF_Door_Lock	3321	Mbed [33]	✓	✓	✗	✓	**FRB09**(CWE-120), **FW29**(CWE-121), **FW38**(CWE-674)
μEMU [55]	3DPrinter †	8046	Arduino [1]	✓	✓	✗	✓	**FP_FW39**
	GPSTracker †	4195	Arduino [1]	✓	✓	✗	✓	**FW30**(CWE-121), **FW31**(CWE-690), **MF02**(CWE-690), **MF03**(CWE-690), **S01**(CWE-690), **S02**(CWE-690)
	utasker [51]_USB	3492	utasker [51]	✓	✓	✗	✓	**FP_FRB06**, **FP_FRB07**, **FP_FRB08**, **FP_FW27**, **FP_FW45**, **FP_MF05**, **MF04**(CWE-1285), **S04**(CWE-123)
	utasker [51]_MODBUS	3781	utasker [51]	✓	✓	✗	✓	**FP_FW40**, **S03**(CWE787)
	Zephyr [41]_SocketCan	5944	Zephyr [41]	✓	✗	✗	✓	**E03**(CWE-120), **FP_FRB11**, **FP_FRB12**, **FP_FW44**, **FW43**(CWE-457), **MF06**(CWE-685), **MF07**(CWE-685), **MF08**(CWE-685), **MF10**(CWE-843), **MF11**(CWE-843), **MF12**(CWE-843), **MF13**(CWE-843), **MF14**(CWE-843), **MF15**(CWE-843), **MF16**(CWE-843), **FP_MF09**, **S05**(CWE-822)
HALUCINATOR [11]	6LoWPAN_Receiver	6978	Contiki [15]	✓	✓	✗	✓	**FP_E04**, **FP_MF18**, **FW36**(CWE-457), **MF17**(CWE-1285)
SPLITS [20]	Contiki [15]_NG_Shell †	4777	Contiki [15]	✓	✓	✗	✓	**S06**(CWE-685)
DICE [?]	MIDI ‡	810	Bare metal	✗	✓	✓	✗	**DI4**(CWE590), **DI5**(CWE590)
	MODBUS ‡	811	FreeRTOS [3]	✗	✓	✓	✓	**DI2**(CWE193), **DI3**(CWE193)
FUZZWARE [44]	Contiki [15]-hello-4-4*	3960	Contiki [15]	✓	✗	✓	✓	**H03**(CWE-131), **H04**(CWE-674), **FW58**(CWE-120), **H06**(CWE-787), **H07**(CWE-787), **H08**(CWE-787), **H09**(CWE-1288), **H10**(CWE-1288)
	Contiki [15]-6lowpan*	3082	Contiki [15]	✓	✗	✓	✓	**HAL02**(CWE-787), **HAL01**(CWE-191)
	Contiki [15]-snmp*	3039	Contiki [15]	✓	✗	✓	✓	**FW59**(CWE-120), **H15**(CWE-125), **H16**(CWE-770), **H17**(CWE-787)
	Zephyr [41]-3330 †	6867	Zephyr [41]	✓	✓	✗	✓	**FW55**(CWE-787)
	Zephyr [41]-bt †	4907	Zephyr [41]	✓	✓	✗	✓	**FW48**(CWE476), **FW47**(CWE787)
	Zephyr [41]-nrf †	4938	Zephyr [41]	✓	✓	✗	✓	**FW54**(CWE-665), **H34**(CWE-476), **H35**(CWE-704), **H33**(CWE-665), **H36**(CWE-697), **H37**(CWE-787), **H38**(CWE-703), **H39**(CWE-703), **H40**(CWE-703), **H42**(CWE-665), **H43**(CWE-476), **H44**(CWE-362), **H45**(CWE-665), **H46**(CWE-457), **H47**(CWE-476), **H48**(CWE-476), **H49**(CWE-703), **H50**(CWE-476), **H51**(CWE-125)
	Zephyr [41]-sam4s †	6954	Zephyr [41]	✓	✓	✗	✓	**FP_FRB16**, **FP_FRB17**, **H52**(CWE-476), **FW53**(CWE-191), **FW50**(CWE-476), **FW49**(CWE-476), **H51**(CWE-191), **H62**(CWE-191)
	Zephyr [41]-sampro †	7176	Zephyr [41]	✓	✓	✗	✓	**FP_FRB18**, **FP_FRB19**, **FP_FRB30**, **FW46**(CWE-787), **FW50**(CWE-476), **FW52**(CWE-476), **FW51**(CWE-191), **FW53**(191), **H59**(CWE-476), **H60**(CWE-476), **H61**(CWE-476)
HOEDUR [46]	Contiki [15]-hello-4-8*	3988	Contiki [15]	✓	✗	✓	✓	**FRB20**(CWE-125), **H03**(CWE-131), **H04**(CWE-674), **FW58**(CWE-120), **H06**(CWE-787), **H07**(CWE-787), **H08**(CWE-787), **H09**(CWE-1288), **H10**(CWE-1288), **H11**(CWE-125), **H12**(CWE-922)
	Contiki [15]-router*	4198	Contiki [15]	✓	✓	✓	✓	**H13**(CWE-476)
	loramac †	5264	Zephyr [41]	✓	✓	✗	✓	**FP_FRB25**, **FP_FRB26**, **H18**(CWE-193), **FP_FRB58**
	gnrc_networking	11912	RIOT [43]	✓	✗	✗	✓	**FP_FRB23**, **FP_FRB31**, **FP_FRB32**, **FP_FRB35**, **FP_FRB36**, **FP_FRB37**, **H19**(CWE-191), **H20**(CWE-476), **H21**(CWE-787), **H22**(CWE-191), **H23**(CWE-191), **H24**(CWE-476), **H25**(CWE-787), **H26**(CWE-252), **H27**(CWE-824)
	Zephyr [41]-f429zi †	5501	Zephyr [41]	✓	✓	✗	✓	**H31**(CWE-415)
MULTIFUZZ [10]	RIOT [43]_CCN_LITE	12675	RIOT [43]	✓	✗	✗	✓	**FP_MF22**, **MF19**(CWE-362), **MF20**(CWE-664), **MF21**(CWE-672), **MF22**(CWE-177)
	RIOT [43]_GNRC	6449	RIOT [43]	✓	✗	✗	✓	**MF23**(CWE-191)
FIRMREBUGGER	Hoverboard*	4440	ChibiOS [48]	✓	✓	✓	✓	**FP_FRB52**, **FRB38**(CWE-252), **FRB39**(CWE-457), **FRB46**(CWE-822)
	Oresat-Control*	20504	ChibiOS [48]	✓	✓	✓	✓	**FP_FRB53**, **FP_FRB54**, **FP_FRB55**, **FRB48**(CWE-685), **FRB49**(CWE-685), **FRB50**(CWE-685), **FRB51**(CWE-685)
	BetaFlight*	26319	Bare metal	✓	✓	✓	✓	**FRB52**(CWE-685), **FRB54**(CWE-685), **FRB55**(CWE-252), **FRB56**(CWE-457)