Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing

Tobias Scharnowski[1], Nils Bars[1], Moritz Schloegel[1], Eric Gustafson[2], Marius Muench[3], Giovanni Vigna[2,4], Christopher Kruegel[2], Thorsten Holz[1] and Ali Abbasi[1]

[1]*Ruhr-Universität Bochum,* [2]*UC Santa Barbara,* [3]*Vrije Universiteit Amsterdam,* [4]*VMware*

Abstract

As embedded devices are becoming more pervasive in our everyday lives, they turn into an attractive target for adversaries. Despite their high value and large attack surface, applying automated testing techniques such as fuzzing is not straightforward for such devices. As fuzz testing firmware on constrained embedded devices is inefficient, state-of-the-art approaches instead opt to run the firmware in an emulator (through a process called *re-hosting*). However, existing approaches either use coarse-grained static models of hardware behavior or require manual effort to re-host the firmware.

We propose a novel combination of lightweight program analysis, re-hosting, and fuzz testing to tackle these challenges. We present the design and implementation of FUZZWARE, a software-only system to fuzz test *unmodified* monolithic firmware in a scalable way. By determining how hardware-generated values are actually used by the firmware logic, FUZZWARE can automatically generate models that help focusing the fuzzing process on mutating the inputs that matter, which drastically improves its effectiveness.

We evaluate our approach on synthetic and real-world targets comprising a total of 19 hardware platforms and 77 firmware images. Compared to state-of-the-art work, FUZZWARE achieves up to 3.25 times the code coverage and our modeling approach reduces the size of the input space by up to 95.5%. The synthetic samples contain 66 unit tests for various hardware interactions, and we find that our approach is the first generic re-hosting solution to automatically pass all of them. FUZZWARE discovered 15 completely new bugs including bugs in targets which were previously analyzed by other works; a total of 12 CVEs were assigned.

1 Introduction

Embedded systems have pervaded our everyday lives, facilitating the transition of our society towards a connected, "smart" world. Security plays an enabling role, and a first step for secure connected devices is to proactively identify their security vulnerabilities in an efficient and scalable way. One way to achieve this goal is automated fuzz testing (*fuzzing*). Unfortunately, fuzzing of embedded devices is challenging [39]. Fuzzing on-device is impractical for firmware due to low fuzzing speeds caused by limited hardware resources [21]. Fuzzing the device in an entirely black-box manner [9] results in missing feedback and limited crash detection, which dramatically limits the fuzzer's effectiveness [21,25,42]. Similarly, fuzzing with the device "in-the-loop" [38,39] also leads to resource constraints due to the need to synchronize hardware and emulated environments.

One way to address the aforementioned inherent issues is *re-hosting*, where firmware is executed in an emulated environment [17, 51]. Various approaches exist to dynamically analyze Unix-based firmware via re-hosting [8, 46, 56], but these approaches do not apply to monolithic firmware, which consists of a single, opaque binary blob. Modern emulation environments [5, 50] allow re-hosting even monolithic firmware by precisely emulating a limited set of hardware. However, such tools require an analyst to find or manually create software equivalents (*models*) of all hardware peripherals for the firmware to run, which is a complex and time-consuming task.

Hence, recent work shifted towards automated hardware-less rehosting, and different strategies to deal with hardware peripherals have emerged. *High-level emulation* approaches attempt to tackle the problem of missing peripheral models by re-implementing and hooking into known libraries to avoid hardware accesses altogether [11, 33, 35]. In contrast, *pattern-based modeling* approaches model a hardware peripheral's registers by matching access patterns to common static hardware register types [18, 22, 23]. A third direction is given by tools deploying *guided symbolic execution*, which treats hardware registers as sources for symbolic inputs. The resulting symbolic values are then solved towards the most promising paths guaranteeing the firmware's operation [7, 57]. All of these strategies allow rehosting of firmware, however, we show that they hit several limitations when the rehosted firmware is tested via fuzzing.

To fill this gap, we propose a fine-grained automated modeling approach, which is optimized for use with a coverage-

guided fuzzer. Our approach is driven by the insight that many accesses to hardware peripherals are short-lived and occur for reasons unrelated to the firmware's overall behavior, such as to check a peripheral's status or set its configuration. For the accesses that *do* influence its behavior, the firmware often leaves large parts of its input unused, e. g., directly by extracting only a couple of bits from a 32-bit value, or indirectly by differentiating between only a handful of status values. By regularly querying more data than it uses, the firmware incurs significant *input overhead* while accessing hardware.

To eliminate this overhead, once per unique peripheral access, we utilize locally-scoped dynamic symbolic execution (DSE) and analyze which parts of the hardware-generated value are actually meaningful to firmware logic. However, unlike prior approaches, we do not use the DSE engine to solve towards specific values for exploring specific parts of the firmware's functionality. Instead, we use the generated constraints to infer generic *access models* geared towards input overhead elimination. These access models are then used to configure an emulator, and their concrete values are later served by the fuzzer. An important aspect of this modeling approach is that at no point during emulation do our models take actual decisions, or prioritize one decision over another. The single goal of this modeling is to present the original set of choices to the fuzzer with as little overhead as possible. Consequently, the fuzzer can still explore all paths that the firmware could take based on hardware-generated values.

We implement our approach in a tool named FUZZWARE and evaluate it against 77 firmware images spanning a total of 19 hardware platforms. Our evaluation shows that while consuming 0.5%-2% of the total experiment computation time, our access models eliminate up to 95.5% of inputs as *input overhead*, allowing the fuzzer to focus on mutating only the relevant 4.5% of hardware-generated values. Compared to state-of-the-art tools [18, 57], FUZZWARE achieves up to 3.25 times the coverage (over a period of 24 hours), discovers additional bugs in samples already analyzed by those tools, and is the first approach to achieve a perfect passing score on the rehosting unit test benchmark introduced by P2IM [18]. Finally, we show how FUZZWARE can be used to identify vulnerabilities in complex, real-world targets. To this end, we analyze the network stacks of two widely-used embedded firmware frameworks, ZEPHYR [55] and CONTIKI-NG [12]. We discovered 15 previously unknown vulnerabilities, leading to the assignment of 12 CVEs.

In summary, we make the following contributions:

- We propose a novel, fine-grained *access modeling* approach which preserves all paths through firmware logic and allows a fuzzer to efficiently mutate only meaningful hardware-generated values.
- We describe and implement FUZZWARE, a highly efficient, self-adapting fuzzing system capable of testing monolithic firmware images in an OS-agnostic way.
- In several experiments, we show that FUZZWARE outperforms prior work on testing embedded firmware. Our prototype found 12 previously unknown vulnerabilities in core embedded network stacks which we responsibly disclosed to the affected vendors.

To foster research on this topic, we will release FUZZWARE, the experimental data sets, and the bug details at `https://github.com/fuzzware-fuzzer/fuzzware`.

2 Technical Background

Before explaining the technical details of our approach, we first discuss different aspects of embedded systems and firmware that make them interesting and difficult to analyze.

2.1 Monolithic Embedded Systems

Embedded systems are often purpose-built, resource-constrained devices. The code these systems run is known as *firmware*. The firmware of an embedded system is responsible for all the device's functions and may or may not contain a traditional OS. Monolithic firmware images, which are the focus of this work, contain none of the traditional metadata found in binary executables. This makes them difficult to analyze by traditional means.

2.2 Memory-mapped IO

Modern CPU architectures allow for accesses to its peripherals via *memory-mapped I/O (MMIO)*. These peripherals are assigned a region of the device's physical memory space. Each of the region's memory locations, termed *MMIO registers* in chip documentation, is accessed via regular load/store instructions. Rather than behaving like normal memory, these instructions instead trigger hardware behaviors in the affected peripheral. For example, consider a button connected to a GPIO pin of an embedded microcontroller (MCU). The MCU can check whether the button is pressed by reading from the MMIO register that represents its GPIO pin. MMIO registers of different types perform certain roles, such as identification, status (e. g., whether a button is pressed), configuration, and data transfer. As such, MMIO registers can *quickly change values* at any time. Registers vary in terms of their size (bit-width) as well as allowing read and/or write operations.

Consider firmware code running on an MCU with a serial port. Figure 1 shows how the I/O of such an MCU may be organized. After configuring the serial baud rate, the firmware waits for a command to initiate a data transfer. The user connects via their own computer to a serial communication port in the MCU. The firmware notices incoming data by checking the serial peripheral's status registers and reads the serial data via the peripheral's data register. Note that there exists no standardized source of input, such as `stdin`. Input into embed-

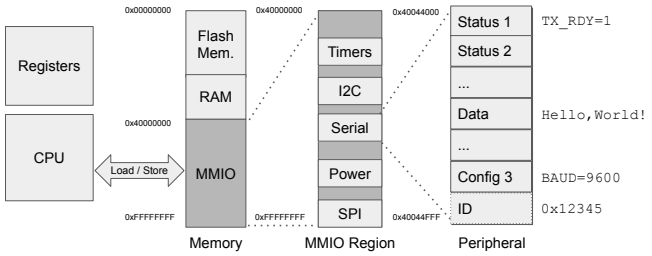

Figure 1: Memory layout of a hypothetical embedded system, showing the correspondence between the memory map, peripherals, and MMIO registers.

ded systems may come from numerous hardware peripherals, potentially even multiple sources in the same device.

2.3 Interrupts and DMA

In addition to the software-initiated communication channels, the hardware has two additional means to communicate with its firmware: First, hardware uses interrupts to notify the software of asynchronous events [32]. For example, a serial port could be configured to trigger an interrupt when data arrives, allowing it to be processed immediately. The CPU tracks these interrupts by their interrupt number and maintains a table of firmware functions, so-called interrupt service routines (ISRs), which process new events. Depending on the CPU model, interrupts can also be selectively disabled, or given a priority level, allowing some interrupts to take precedence. The association of the peripheral with its interrupt number depends on the specific CPU model in use and may vary widely, even within products from the same vendor. The second asynchronous communication channel is called direct memory access (DMA), which is configured via MMIO. Using DMA, a peripheral is able to update firmware-accessible regular memory by talking directly to the memory controller and without involving the CPU. While interrupts are universally used as a source of input into firmware, DMA is primarily used in high-throughput scenarios such as USB.

2.4 Re-Hosting Embedded Systems

Firmware re-hosting is a way to run a firmware binary image without relying on actual hardware. Emulating firmware in a fully virtualized environment allows multiple emulator instances to be run in parallel and thus enables effective dynamic analysis techniques such as fuzzing. Generally speaking, to re-host firmware, one needs to emulate three main interactions between firmware and hardware: interrupts, DMA, and MMIO. From these three, MMIO, which we focus on in this work, represents a significant share and is used universally. We need to properly handle MMIO accesses to even reach the parts of firmware that perform DMA. To handle MMIO behavior in firmware, various approaches take different directions. For example, QEMU fully re-implements the behavior of each MMIO register. While this approach can precisely emulate MMIO, it requires a significant amount of engineering effort for each emulated peripheral, as well as access to full hardware documentation.

An alternative approach to modeling MMIO peripheral behavior is *approximation*. The basic idea is to involve a fuzzer to handle MMIO accesses just as they would occur in practice. In its most naïve form, a fuzzer-provided value can directly be served as a hardware-generated value, whenever firmware code accesses an MMIO register. This general approach is appealing, as it allows running the firmware without a priori knowledge about MMIO usage and handling MMIO accesses even if no precise implementation of a peripheral is available. However, as we will discuss next, this is very challenging: a fuzzer has to provide inputs for an overwhelming amount of MMIO accesses, many of which are irrelevant to firmware behavior, and hence such an approach does not scale to real-world systems.

3 MMIO Access Handling

As discussed in the previous section, firmware universally relies on MMIO accesses. Therefore, handling MMIO accesses during emulation is crucial to enabling efficient firmware fuzzing. In the following, we investigate why a naïve fuzzing-based approach to MMIO access handling exposes the fuzzer to large amounts of *input overhead*. Next, we discuss previous approaches to removing this overhead via MMIO modeling and their shortcomings in enabling effective, scalable fuzzing.

3.1 Input Overhead

Assume a naïve approach where bits from a random byte stream generated by a fuzzer are served as *hardware-generated values* (i. e., values which, from the firmware's perspective, are provided by a hardware MMIO register). We refer to these bits as the fuzzer-mutated *input space* which is then processed by the firmware logic. This input space contains both *relevant* bits, i. e., bits affecting the firmware logic, and *input overhead*. For each MMIO access, we differentiate between two types of input overhead:

- *Full input overhead*: No bit provided by the fuzzer is relevant. In other words, the emulator could have handled the MMIO access statically, e. g., by providing an arbitrary value.
- *Partial input overhead*: One or more bits are relevant, i. e., they influence the firmware logic (e.g., by influencing control-flow decisions), while other bits do not. For example, consider firmware code that accesses a 32 bit wide MMIO register, but actually uses only 8 bits of the resulting hardware-generated value. If the full 32 bits of fuzzing input are consumed to serve the access, 24 bits of *partial input overhead* are introduced.

```
u8 serial_getc() {
    // Busy-check for data presence
    while (mmio->status != HAS_DATA){}; (1)
    // Indicate read via GPIO
    gpio->val = gpio->val | UART_ACTIVE; (2)
    // Read full data register
    u32 data = mmio->data; (3)
    // Mask data part and return
    return data & 0xff;
}
```

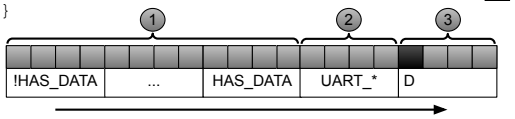

Figure 2: An example of a function for retrieving input from a serial port peripheral. The annotations indicate resulting MMIO accesses relating to overhead (gray) and actual application data (black).

```
1  void perform_op() {
2   // Check requested operation
3   switch (mmio->op) {
4     case A: handle_A(); break;
5     case B: handle_B(); break;
6     case C:
7       if(mmio->status == SPECIAL) {
8         handle_C_special(); break;
9       } else {
10        handle_C_default(); break;
11      }
12    default: housekeeping();
13  }
14 }
```

Figure 3: An example of a function that takes actions based on MMIO input using switch/case and if/else constructs.

While fundamentally simple, these overhead types govern the fuzzer's efficiency: Exposing the fuzzer to input overhead leads the fuzzer to mutate bits that do not affect firmware logic, hence wasting resources. To better understand this in practice, we explore two code examples which are inspired by real-world firmware and represent typical firmware operations.

Example 1. Figure 2 shows a typical firmware function that retrieves a character from a serial port. This function waits for the serial port to have data available (1 in Figure 2), triggers a GPIO write (e. g., to turn on a busy indicator LED) in 2, and finally returns one byte of data. The waiting for serial data involves polling for a specific value (1), defined by the hardware, which indicates that one byte has arrived. Without modeling, the fuzzer is rather unlikely to feed the correct value to the MMIO access, thus bottle-necking on the loop until the correct input is found by chance. As only one specific value is accepted, this is a prime example of full input overhead hindering the fuzzing process. While writing to GPIO might be seen as an MMIO write operation, GPIO bits are typically packed into registers with 32 bits representing 32 GPIO pins. Therefore, to perform a GPIO operation without affecting the nearby bits, we must read 4 bytes (2), flip the desired bit, and write the result back. The data initially read has no impact on the program (full input overhead). Eventually, we read the actual data from the serial port (3). While this serial port is byte-oriented, the MMIO register itself is typically 4 bytes wide, i. e., we read 3 bytes more than needed. To prevent any side-effects of this operation, the firmware masks off only the data byte and returns it. This is a case of partial input overhead. As a result, *only a single one* of the 12 (or more) bytes read in this function is passed on to firmware logic (marked as a black square in Figure 2). For this function, a naïve modeling approach that passes fuzzer input to each MMIO access has a minimum input overhead of 92%. The actual overhead is likely even larger if the fuzzer needs multiple attempts to guess the value of `HAS_DATA`.

Example 2. Figure 3 shows another set of typical firmware code constructs that introduce a less obvious source of *partial input overhead*. The function decides which operation to execute based on a hardware-generated value (Line 3) and, in one case (Line 7), also checks the peripheral's status register. Without further insight, the fuzzer would have to provide 4 bytes (32 bits) for each MMIO access and correctly guess meaningful values. The fuzzer's large input space is contrasted by the limited number of meaningful values it can find: The firmware differentiates between only 2 status conditions (special or non-special) as well as 4 different operations (A, B, C, or default). These choices can be expressed by only 1 and 2 meaningful bits respectively, resulting in 94% and 97% *partial input overhead*.

3.2 Previous MMIO Modeling Approaches

In essence, recently proposed hardware-less rehosting approaches deploy one of the following strategies to deal with unknown peripherals:

- *High-level emulation* gets past the need of modeling specific hardware peripherals by completely avoiding MMIO accesses. Previous work abstracts firmware code that performs low-level MMIO accesses by hooking into, and manually handling, higher-level library functions [11, 35].

- *Pattern-based MMIO modeling* tackles MMIO accesses directly. They allow emulated firmware to perform MMIO accesses and attempt to reduce the input space by using access pattern-based heuristics [18, 22, 23]. This means that one observes accesses to an MMIO register, matches these observations to common, pre-defined patterns, and assigns a model to that specific MMIO register. This model then determines how to serve future accesses to this register.

- *Guided Symbolic Execution-based modeling* approaches [7, 57] improve upon pattern-based MMIO modeling. Instead of assigning static patterns based on heuristics, accesses to hardware are treated as symbolic values. Whenever a concrete value for one MMIO access is needed, the underlying symbolic variable is

solved towards the most *promising* path, i. e., more coverage of the firmware logic.

We identify three problems with the current approaches to MMIO modeling: (1) per-firmware manual effort, (2) incomplete overhead elimination, and (3) path elimination.

Per-firmware manual effort. All prior solutions require manual work when preparing specific firmware for fuzzing campaigns. For instance, this includes the creation of HAL abstractions, correcting misclassified MMIO registers, or identifying *alive* and *kill* points to steer the symbolic execution engine. Although recent approaches [7, 57] deploy heuristics to reduce the manual involvement, we note that in practical usage, firmware-specific knowledge is still required, limiting the flexibility for fuzz testing.

Incomplete overhead elimination. While effective in removing full input overhead, pattern-based approaches generally make assumptions about hardware behavior based on conventions of how firmware is "typically" implemented rather than considering actual firmware logic. However, they are unable to identify which parts of an input are actually used by firmware, i. e., they cannot eliminate *partial input overheads*.

Consider Example 1 from the perspective of pattern-based register modeling. As pattern-based MMIO modeling approaches lack insight into firmware-internal logic, they are unaware of the fact that 3 out of the 4 bytes read from the serial data register (③ in Figure 2) are discarded. As a result, these approaches cannot eliminate the 75% of partial input overhead introduced by the access.

While incomplete overhead elimination does not affect rehosting itself, it becomes problematic during fuzz testing: a fuzzer will spend a significant amount of time mutating values that have no impact on program logic.

Path elimination. While guided symbolic execution-based approaches reduce large parts of input overhead, they willingly accept to leave specific parts of the firmware unexplored during fuzz testing, i. e., eliminating available execution paths from the firmware. High-level emulation replaces full parts of the firmware with abstractions, and pattern-based MMIO may miscategorize certain registers or wrongly conclude that no relevant options exist for a given MMIO access. Guided symbolic execution-based approaches use heuristics and human insights to decide which paths are worthwhile to explore.

Although path elimination allows for rehosting of the firmware, it has severe consequences for fuzz testing. First, eliminating specific paths may render large parts of the firmware's functionality unreachable and in turn impossible to analyze. Even assuming correct modeling, path elimination will affect error handling and recovery functions, which may contain bugs, and should not be dismissed. Furthermore, we argue that differentiating between regular firmware behavior and error handling functionality is an undecidable problem in the general case: Error conditions may be met in firmware logic with complex diagnostics and recovery attempts. At the same time, regular firmware behavior that inconspicuously waits for asynchronous events may appear as an infinite loop which does not perform any meaningful operations. This directly reflects on state-of-the-art solutions, which run into execution stalls without human assistance [18, 57].

Following these insights, we conclude that an effective rehosting solution for fuzz testing must avoid path elimination, while at the same time reducing the per-firmware manual effort and eliminate as much input overhead as possible, which is directly reflected in our design.

4 Design

In the following, we introduce the design of FUZZWARE, a generic firmware fuzzing approach that allows a fuzzer to efficiently explore firmware behavior by precisely eliminating both partial and full input overhead.

To this end, we base our modeling on lightweight program analysis techniques that allow us to spot partial uses of hardware-generated values. To analyze the behavior of firmware code, we employ dynamic symbolic execution (DSE) [43]. DSE allows us to generate a set of constraints representing all possible uses of a hardware-generated value. Evaluating these constraints allows us to narrow down the set of values to be explored by the fuzzer. Typically, using symbolic execution for modeling introduces high computation costs due to the state explosion problem. We avoid this drawback by using *local DSE*, where DSE is used only to execute the code in the context of a specific MMIO access. We describe the details of limiting the DSE's scope in Section 4.3.

4.1 Prerequisites and Threat Model

FUZZWARE has the following prerequisites and threat model:

Prerequisites. FUZZWARE shares two basic prerequisites with all other re-hosting systems: First, we assume that we are able to obtain a binary firmware image for the target device. Second, just like other re-hosting systems, we assume basic memory mappings such as RAM ranges and the broad MMIO space to be provided. Depending on the target CPU architecture, these generic ranges may be standardized [1].

Threat Model. Given no additional knowledge about the specific hardware environment of a given binary firmware image, we assume during fuzzing that an attacker is able to control the inputs provided to the firmware. Commonly, these inputs may correspond to the contents of an incoming network packet read via MMIO, data received via a serial interface, or sensory data such as temperature measurements. We analyze situations where an attacker has less control over hardware-generated values in Section 6.4.

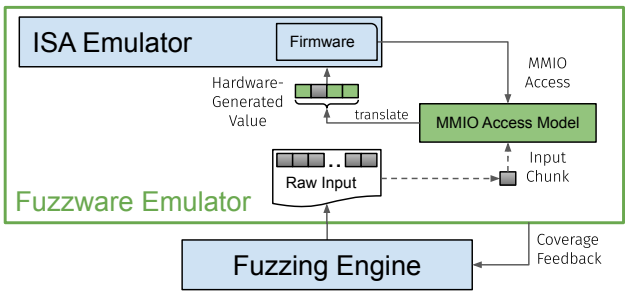

Figure 4: FUZZWARE's MMIO access handling design. The fuzzing engine generates a raw input file. Upon MMIO accesses, chunks of the input file are consumed by MMIO access models and translated into (potentially larger) hardware-generated values, which are then served to the emulated firmware. Once the raw input is exhausted, coverage feedback is provided to the fuzzing engine to guide the fuzzing process.

4.2 FUZZWARE's Emulator

We now describe the design of FUZZWARE's emulation component. Figure 4 shows that, from a high-level point of view, FUZZWARE uses an ISA emulator and a coverage-guided fuzzing engine (fuzzer). As FUZZWARE aims to eliminate *partial input overhead*, we introduce *access models*, a mechanism to *translate* small amounts of fuzzing input bits into values that are meaningful to firmware logic, while eliminating input overhead in the process.

We start by loading a given monolithic firmware image into the ISA emulator. We set up a harness that dynamically intercepts all *MMIO accesses*, i. e., memory operations performed by the emulated firmware code on all addresses within MMIO regions. The harness is provided a *raw input* (i. e., a plain binary file) generated by the coverage-guided fuzzer; then, it starts firmware code emulation. The raw input is consumed in chunks to serve MMIO accesses. Whenever firmware code performs an MMIO access, the harness checks whether we already assigned an *MMIO access model* to this specific access. If a model is available, and depending on the type of input overhead, the harness may be able to handle the access without consuming any raw input (in case of *full input overhead*). Otherwise, the harness consumes a chunk of raw input and translates it into a hardware-generated value via the model (*partial input overhead*). The hardware-generated value is then used to serve the MMIO access. The emulator runs the firmware code until the fuzzer's raw input is exhausted and it can no longer serve MMIO accesses. We term this emulation cycle an *emulation run*. As an emulation run is concluded, the harness restores firmware to its clean state, and reports coverage feedback for the previous emulation run to the fuzzer. Based on this feedback, the fuzzer generates another raw input and provides it to the harness for the next emulation run.

However, if during emulation a specific MMIO access has no model assigned yet, raw input chunks are used as hardware-generated values without translation. In parallel to ongoing fuzzing, FUZZWARE initiates modeling of each newly seen *MMIO access context* (the pair of current program counter and MMIO address). In a separate emulator instance, we create a snapshot of the firmware's state (i. e., register and memory) right before the MMIO access. We use symbolic execution from this snapshot to derive a matching model (described in detail in Section 4.3). We then re-configure the emulator with new models, allowing the fuzzer to more effectively discover further firmware logic with less input overhead.

We bootstrap this fuzzing loop by providing no initial MMIO access models. Models are continuously generated and added to the emulator configuration while the fuzzer is active. This design provides a generic, self-adapting firmware emulation environment, which allows a fuzzing engine to explore unknown firmware with minimized input overhead.

4.3 Modeling Approach

As previously explained, for each MMIO access context (i.e., program counter and accessed MMIO address), we construct an access model. To do so, we replay the input for which the new MMIO access is performed, and snapshot the emulator's register and memory state just before firmware would perform the MMIO access. We pass this snapshot on to our DSE engine for modeling, and symbolically execute the code, starting from the snapshot. Each MMIO access observed during the symbolic execution is treated as a separate symbolic variable.

Modeling Analysis Scope. We track the first MMIO access (as well as any additional accesses from the same access context), to follow whether the resulting symbolic variable is still *alive*, i. e., at least one symbolic expression in memory or a register still depends on it. The symbolic execution continues until one of the following events occurs:

1. All tracked symbolic variable are dead (i. e., not alive),
2. the current function returns,
3. a tracked symbolic variable is leaving the scope of the analysis (i. e., it is written to global memory or to a stack frame of a function higher in the call stack), or
4. a pre-defined limit of computation resources is exhausted (timeout, number of symbolic states, or number of DSE steps was reached).

Using these exit conditions, we narrow down DSE to a small, manageable scope, in which we are able to observe all actions that firmware takes based on an MMIO access. At the same time, we do not model uses of a hardware-generated value beyond this scope. The rationale behind this scoping decision lies in the short-lived nature of MMIO register states (see Section 2.2), which forces firmware to frequently access and quickly discard hardware-generated values. As we will show in Section 6.1, our evaluation supports this notion.

Upon hitting one of the exit conditions, the modeling logic analyzes the resulting symbolic states. Each symbolic state corresponds to a possible path that firmware code could take depending on hardware-generated values. A symbolic state has a set of different path constraints, i. e., conditions that

Table 1: MMIO Access Models. HW denotes the Hamming Weight.

Model Type	Overhead	Parameters	# Fuzzing Bits	Output
Constant	full	constant	-	constant value
Passthrough	full	-	-	stored value
Bitextract	partial	bitmask	HW(bitmask)	filled bitmask
Set	partial	constants	$log_2\|\text{constants}\|$	selected value
Identity	none	-	full access size	fuzzing bits

hardware-generated values need to adhere to, as well as possible symbolic expressions containing tracked variables which are still alive. These symbolic states are then used as input to assign and configure a model for the analyzed MMIO access.

Model Design Considerations. Based on the previous discussions, two aspects are central to our model design: First, models provide *reproducible* translations. Performing an emulation run for a given raw input multiple times has to result in identical firmware executions. We require identical behavior as we generate MMIO access snapshots for modeling in separate emulator instances, in parallel to ongoing fuzzing. To keep translations reproducible, we derive hardware-generated values exclusively from chunks of fuzzing input. Second, we design our models to *preserve firmware code paths*. While we aim to eliminate as much input overhead as possible, we conservatively apply models that do not make firmware code paths unavailable in the process. Thus, we only model accesses based on variable uses that we can fully observe. In case a live variable leaves our analysis scope, we base our modeling on the constraints that firmware logic has already placed on the modeled variable (e. g., a bit mask has been applied before data gets returned, see (3) in Figure 2).

Error Handling and Execution Stalls. Naturally, by preserving all firmware code paths, we also allow the fuzzer to exercise error paths. This is intentional: Contrary to previous modeling approaches, we explicitly do not try to prioritize specific paths or remove entire paths that *appear* uninteresting (cf. Section 3.2). Instead, we find that code which handles irregular conditions may contain bugs, and should likewise be included in the analysis. This inevitably leads to inputs that result in stalled firmware execution. However, we note that these cases are seamlessly dealt with by the fuzzer: Whenever execution is stalled, the fuzzer will recognize missing code coverage. Consequently, the corresponding inputs will be discarded as uninteresting, and the mutation engine will quickly yield inputs with more significant code coverage. We want to stress that this conscious decision to explore error handling does not only allow for discovery of bugs which may be missed otherwise, but also enables truly robust and automated fuzzing of firmware, as neither a human analyst nor heuristics are needed to identify *interesting* paths.

4.4 FUZZWARE Model Definitions

We define a total of five generic MMIO access models that can be assigned from a DSE-produced set of symbolic states. Each of these models provides a blueprint to the emulator for how to handle a specific MMIO access and systematically remove input overhead, full or partial, either for typical control-flow based MMIO uses (i. e., taking different actions based on a value) or data-based MMIO uses (i. e., reading data and dismissing all or parts of it). Some of these generic models accept parameters by specification. We use the symbolic states to first assign a generic model and then instantiate it via parameters for the given MMIO access. The generic models contain a specification for the emulator on how to apply the model parameters to determine a hardware-generated value. For models handling *full input overhead*, model parameters alone are sufficient to handle the access, without consuming any fuzzing input. For models handling *partial input overhead*, the emulator requires fuzzing input to apply the model. In these cases, it uses the model's parameters to *translate* a fuzzing input chunk into a hardware-generated value.

We now detail our five generic models. For each one, Table 1 shows which type of input overhead it handles (Overhead), which parameters it uses (Parameters), how many raw fuzzing bits an access consumes (# Fuzzing Bits), and how models use parameters to translate the raw fuzzing input into the hardware-generated value (Output).

1) Constant Model. This model describes MMIO accesses where a specific constant is used as part of a comparison, which must be satisfied to allow execution to proceed (see (1) in Figure 2).

2) Passthrough Model. This model is assigned to accesses where the hardware-generated value is determined to not affect the firmware's state. We treat the MMIO access like a regular memory access. These include, for example, accesses to configuration registers (see (2) in Figure 2).

3) Bitextract Model. The bitextract model is used when only a portion of the bits read from MMIO are used by the firmware. For example, this is the case when four bytes are read from an MMIO register and a bitmask is applied to only retain a few bits while the others are discarded (see (3) in Figure 2). Note that similar effects occur for bit shifts, truncations, and equivalent instruction composites.

Examples: A 4 byte-wide MMIO access is performed with a model-computed bit mask of 0x00ff0000. The emulator consumes a byte of fuzzing input, e. g., 0x4e. The emulator serves 0x004e0000 as the hardware-generated value for the MMIO access. For a bit mask of 0xfff0000f and a consumed raw input chunk 0xabf8, the emulator serves 0xabf00008.

4) Set Model. The set model handles the situation where a (part of the) hardware-generated value is checked against different values to determine control flow. The model is applied in case a discrete list of values can be precomputed such that each value represents exactly one of the possible control-flow options. A chunk of raw fuzzing input is interpreted as the fuzzer's choice from among the different options for each individual access. Possible instances include status and iden-

tification registers, where the firmware performs different actions based on the hardware-generated value (see Figure 3).

Example: A 2 byte-wide MMIO access is performed for a model-computed list of four precomputed values [1,5,7,128]. The emulator consumes 2 bits of fuzzing input, e. g., 0x1. The emulator serves 0x0005 as the hardware-generated value.

5) Identity Model. This model is assigned if DSE determines that all bits of a hardware-generated value are meaningful (i. e., used by firmware). It is also used as a fallback in case an unconstrained symbolic variable escapes the analysis scope, or if DSE does not complete within its resource limits. In these situations, we conservatively assume that every bit of the hardware-generated value may later be used by the firmware. Thus, we allow the fuzzer to try all values and therefore to discover all firmware paths. As we will show in Section 6.1, this fallback is rarely required in practice.

Model Computation By Example. For further explanation, we re-visit the busy check of the serial peripheral's status shown in (1) in Figure 2. While stepping through the loop, our symbolic execution reaches the comparison and splits the execution into two states—one which exits the loop and another which takes an additional loop iteration. By generating multiple of these states and inspecting path constraints, we can show that for each state that exits the loop, the hardware-generated value has to be equal to `HAS_DATA` during the last access, while prior accesses had to be different from `HAS_DATA`. Consequently, firmware execution does not continue without the hardware-generated value being equal to `HAS_DATA`. We can use this information to assign the *Constant Model*, parameterized with the value of `HAS_DATA`. Similarly, for the GPIO update in (2), DSE will show that the queried hardware-generated value is only written back to an MMIO address, but not involved in a constraint on the execution state otherwise. Hence, we assign the *Passthrough Model*. Finally, for (3), DSE shows that while no constraints exist on the path itself, a masked part of the hardware-generated value is returned from the function. As the DSE terminates on the function boundary (to avoid path explosion), we assign a *Bitextract Model*.

4.5 Interrupt, Timer and DMA Handling.

As described in Section 2.1, interrupts are an asynchronous source of input into firmware logic. As an ISA emulator does not contain any notion of peripherals to raise interrupts, this behavior has to be triggered by FUZZWARE. Per default, FUZZWARE raises each of the currently-enabled interrupts in a rolling fashion after a certain number of basic blocks is executed. The set of enabled interrupts is tracked by examining the state of the CPU's interrupt controller during execution.

Among other peripheral behavior, FUZZWARE mimics interrupt-based timer peripherals this way. To provide additional flexibility in exploring how firmware logic reacts to specifically-timed events, FUZZWARE allows precise control over both when and which interrupts should be raised. Similar to how fuzzing input is used by access models to determine hardware-generated values, fuzzing input can be used to determine the timing of the next interrupt, as well as its number. This allows the fuzzer to discover the influence of very specific interrupt timings on firmware behavior.

Note that FUZZWARE can support some forms of DMA by defining transfer buffers as MMIO regions. However, FUZZWARE currently does not explicitly model DMA in an automated way given that this is out of scope for this work.

5 Implementation

We implement a prototype of FUZZWARE targeting the ARM Cortex-M standard. We chose this platform due to its wide adoption in practice and projected future popularity [4]. The implementation is designed such that support for other targeted ISAs is possible in the future.

5.1 FUZZWARE's Emulator

Our implementation is based on Unicorn Engine [49] as the base ISA emulator and we use legacy AFL [54] as the fuzzing engine for a fair comparison with other modeling approaches. We also integrated AFL++ [19] for its extended feature set and baseline performance. We handle MMIO accesses by registering memory access hooks for MMIO regions with the native Unicorn API. We handle hooked read accesses by writing the output of the assigned model (as described in Section 4.3) to the accessed MMIO address before the read operation is performed. We associate a memory access with its corresponding model by its *MMIO access context*, i. e., the pair of program counter and MMIO address. If no associated model exists, we default to handling the access according to the *Identity Model*. We use three generic files as initial fuzzing inputs (each 512 bytes in size): All zero-bits, all one-bits, and concatenated 32-bit values with a shifting 1-bit each.

Empirically, we have found that consuming raw fuzzing input provided by an unmodified byte-oriented fuzzer on a bit-granular level conflicts with the heuristics that drive the fuzzer's input mutation process. Consequently, to handle MMIO accesses, we consume raw inputs on a byte-granular level. For example, while each access to a set model with four elements requires a minimum of two bits of fuzzing input, a byte is consumed in our current implementation.

Timers and Interrupts. Timers and interrupts are a source of nondeterminism in firmware execution. As discussed in Section 4.3, we require emulation runs to be fully reproducible. To achieve precisely reproducible timing behavior, we measure elapsed time by the number of emulated basic blocks. We also extended the Unicorn Engine with an implementation of the interrupt controller (NVIC) and the system tick timer (SysTick), which are defined in the Cortex-M standard.

5.2 MMIO Access Modeling

For DSE, we chose angr [44, 45] as an engine, as it—just like Unicorn Engine—readily supports a wide range of ISAs and lends itself well to including more architectures.

After loading a firmware state snapshot into angr and creating a symbolic variable for the hardware-generated value representing the tracked MMIO access, we track the variable's liveness via reference counting. We increment the reference count whenever DSE writes a symbolic expression containing the tracked variable to a register or to memory, and we decrement the count whenever such an expression gets overwritten. To track whether register value assignments from a concrete restored state snapshot influence modeling results, we taint registers after loading the snapshot.

Upon hitting an exit condition as described in Section 4.3, we check the live symbolic expressions and constraints on the resulting states for adherence to each model definition as detailed in the following.

1) Constant Model: All tracked variables are no longer referenced, but constrain the resulting states. A single common value v for the latest tracked variable exists between all resulting states with the following property: For any previous-to-last variable, assigning v does not satisfy the state constraints. The constant value v parameterizes the model.

2) Passthrough Model: All tracked variables are no longer referenced and do not constrain any of the resulting states.

3) Bitextract Model: All state constraints and symbolic expressions remain unchanged after a bit mask has been applied to each tracked symbolic variable in each state. The bit mask with the lowest Hamming weight parameterizes the model.

4) Set Model: All variables are no longer referenced, but constrain the resulting states. For each state and reference-counted variable, a value can be found that does not satisfy the path constraints of any of the other states. In other words, the sets of constraints on each path form partitions of the input space between states. The minimum representative of each partition is chosen as a value in the configured set, which parameterizes the model.

5) Identity Model: None of the above models apply, or no model was found within DSE limits.

If multiple models apply, the one with the highest reduction of input overhead is chosen. As the limit for the DSE computation, we set the default run time to 5 minutes per model and a maximum of 1,000 symbolically executed basic blocks, which we have found to work well in practice.

6 Evaluation

We evaluate FUZZWARE by considering the following research questions:

RQ 1 How computationally expensive is the implemented symbolic execution-based modeling?

RQ 2 How many optimized modeling opportunities does FUZZWARE miss due to its conservative scoping?

RQ 3 Are FUZZWARE's MMIO access models applicable to a wide variety of firmware and hardware platforms?

RQ 4 How does FUZZWARE perform compared to previous methods in fuzzing monolithic firmware?

RQ 5 Can FUZZWARE be used to uncover previously unknown bugs in real-world firmware?

To answer these questions, we performed different experiments, targeting 77 different firmware images for 19 different hardware platforms, summarized in Table 4 in the appendix. First, we quantified the amount of input overhead that access modeling eliminates and studied how this translates into code coverage. Second, we applied FUZZWARE to a set of real-world firmware samples used in concurrent work. Third, we used FUZZWARE to test network stacks of widely-used embedded firmware frameworks with the goal of uncovering network packet processing bugs. Finally, we analyzed the root causes of the crashing test cases produced by FUZZWARE.

6.1 Access Modeling for Fuzzing

In a first step, we focus on the costs and the general applicability of FUZZWARE's access modeling on the fuzzing-based firmware exploration process (**RQ 1**, **RQ 2**, and **RQ 3**).

For the initial part of our evaluation, we use two sets of firmware targets: First, we created a unified application-level program from which we generate firmware images for ten hardware platforms supported by ARM's Mbed OS [36]. We use a unified application as from a modeling point of view (and probably counter-intuitively), compiling the same application-level program for 10 different boards will look vastly different, while compiling 10 different application-level programs for the same board will effectively look the same to MMIO modeling. This is why we reach diversity by compiling the same program for 10 boards. To expand on the application-level diversity, we also test FUZZWARE on the 66[1] unit tests originally published by the authors of P2IM [18].

Our test application repeatedly triggers hardware platform-specific driver behavior by calling different high-level Mbed OS APIs, which then resolve to its platform-specific driver functions and thus trigger MMIO accesses. The test application then prompts the user for a password over the serial port. If the correct password was entered, the firmware exposes a vulnerable function accepting input from the serial port. We use this application to repeatedly trigger the underlying hardware-specific driver implementations for each platform.

To provide the baseline data for our evaluation, we fuzz each of the ten Mbed OS targets for 24 hours, once with

[1] Originally, this data set consisted of 70 firmware unit tests [18], but a recent errata removed four of them for validity reasons.

MMIO access modeling enabled and one time with MMIO access modeling disabled. We repeated this experiment ten times to account for the fuzzer's inherent nondeterminism as recommended by Klees et al. [29]. We used a 40-core Intel Xeon Gold 6230 CPU @ 2.10 GHz machine running Ubuntu 18.04.4 LTS and assigned each FUZZWARE instance two CPU cores. We visualize the time spent to discover the individual characters of the password in Figure 6 in the appendix. Based on these experiments, we collected additional metrics on several aspects of FUZZWARE's operation.

Costs of Model Generation. To evaluate the (one time) computation costs incurred by our modeling (**RQ 1**), we collected the start and completion timings of all model generation jobs. On average, 62 models have been generated during a 24-hour experiment for a single firmware image, which took an average of 6.34 minutes (6 seconds per model) to compute.

Input Overhead Elimination. After analyzing the costs incurred by modeling, we quantify its overall elimination of input overhead. Table 2 shows how much input overhead (M) different models eliminated and how much fuzzing input (F) they consumed. As described in Section 5, the current implementation of access models operates with byte granularity. Every second row shows the input overhead reduction for an assumed bit-granular model implementation. The resulting data shows that in total, the current implementation eliminates a minimum of 49.3% and a maximum 83.4% of the input space (in ARCH_PRO and NUCLEO_L152RE, respectively). When considering a bit-granular implementation, these values increase to 49.7% and 95.5%, respectively. Over all runs, the input space was reduced by nearly 80% and could have been reduced by nearly 90% with bit-granularity. We can also see that depending on the target, input overhead differs significantly. It is worth mentioning that depending on the firmware sample, some model types simply do not apply. If the bit-widths of MMIO accesses exactly match the amounts of data actually used within firmware code, Bitextract optimizations are not required (see ARCH_PRO).

To determine the opportunities of input overhead reduction that FUZZWARE might have missed (**RQ 2**), we re-visited the cases where an Identity model was assigned, meaning that no input overhead was eliminated for the given MMIO accesses. In total, of the 623 unique models that were generated during the experiment, 34 have been assigned the Identity Model. We manually verified that in 19 instances, full values were used within firmware logic, leaving no room for overhead reduction. The remaining 15 cases (less than 2.5% of the 623 models) involved processing where DSE resource limits applied. In these cases, modeling conservatively assigned an Identity model and fell back to allowing the fuzzer to try all values. This ensures that we are not assigning a wrong model which could hide parts of firmware code from the analysis. Only rarely encountering this fallback is expected: firmware typically processes a hardware-generated value immediately after reading it, as MMIO register states are short-lived, and may quickly change values.

Table 2: Percentage of fuzzing input (F) used and overhead reduction (M) achieved, per model type. We analyze how much fuzzing input each model consumes (if the model consumes any) and how much input overhead each model eliminates (the Identity model does not eliminate input overhead). *(CN: Constant, PT: Passthrough, ID: Identity)*

Target		CN	PT	Set		Bitextract		ID	Total	
		M	M	M	F	M	F	F	M	F
ARCH_PRO	bytes	45.8	3.3	0.1	0.5	0	0	50.3	**49.3**	50.7
	bits			0.6	0.1				**49.7**	50.3
EFM32GG_STK3700		45.2	0.4	0.3	0.1	33.8	11.3	8.8	79.8	20.2
				0.4	0.0	33.9	11.3		79.9	20.1
EFM32LG_STK3600		46.3	0.4	0.3	0.1	34.7	11.6	6.6	81.7	18.3
				0.4	0.0	34.7	11.5		81.9	18.1
LPC1549		48.6	1.6	0.2	0.1	0	0	49.6	50.4	49.6
				0.3	0.0				50.4	49.6
LPC1768		46.6	3.3	0.1	0.5	0	0	49.4	50.1	49.9
				0.6	0.1				50.5	49.5
MOTE_L152RC		34.4	0.5	20.0	6.7	28.4	9.5	0.5	83.3	16.7
				25.9	0.8	34.3	3.6		95.1	4.9
NUCLEO_F103RB		32.3	0.7	20.8	6.9	28.6	9.5	1.3	82.2	17.8
				26.8	0.9	34.9	3.2		94.7	5.3
NUCLEO_F207ZG		25.7	1.0	22.2	7.4	29.2	13.5	1.1	78.1	21.9
				28.7	0.9	38.5	4.2		93.8	6.2
NUCLEO_L152RE		34.4	0.6	20.4	6.8	28.0	9.3	0.5	**83.4**	16.6
				26.4	0.9	34.2	3.2		**95.5**	4.5
UBLOX_C027		43.6	8.2	0.1	0.4	0	0	47.7	51.9	48.1
				0.5	0.0				52.3	47.7
Total	bytes	**35.9**	**0.8**	**15.6**	**5.2**	**27.1**	**9.5**	**5.9**	**79.4**	**20.6**
	bits			**20.2**	**0.7**	**32.0**	**4.5**		**89.0**	**11.0**

MMIO Access Model Generality. The authors of P2IM published a set of 46 firmware images comprising 66 unit test cases. These are designed to test the ability of an emulation system to deal with diverse types of common hardware peripherals on different combinations of firmware and hardware platforms (eight hardware peripherals, three MCUs, and three OS libraries), as well as interrupt-based and synchronous input passing mechanisms. For these 66 test cases, previous work achieves passing rates of 83% and 95%[2], respectively [18, 57]. Regarding **RQ 3**, we reproduced these test cases by running FUZZWARE for 10 minutes. FUZZWARE passed *all* of the 66 test cases. Consequently, FUZZWARE is the first automated and generic emulation system to pass 100% of the P2IM unit test cases, demonstrating the robustness of its models, its general applicability, and the advantage of approaches not relying on path elimination.

6.2 Comparison with the State of the Art

To assess FUZZWARE's efficacy, we compare it with μEMU [57] and P2IM [18], two state-of-the-art tools for hardware-less re-hosting. Like FUZZWARE, they support generic monolithic firmware fuzzing without significant manual intervention. As the evaluation data set, we use the 21 real-world firmware samples presented in μEMU, which includes 10 samples previously tested by P2IM.

For each sample, we performed five 24-hour fuzzing iterations for each target on virtualized dual core machines running on Intel Xeon Silver 4114 CPUs at 2.20 GHz on Ubuntu

[2] Based on the remaining 66 of 70 unit tests, adjusted from 79% and 93%.

Figure 5: Uniquely discovered basic blocks for P2IM real-world firmware samples over five 24h runs. Displayed are the median number of discovered basic blocks, alongside with minima and maxima over the individual runs.

18.04.4 LTS. For all experiments, we re-used exact input seeds where provided, and used the configurations published alongside the tools where applicable. For FUZZWARE, we always use its default configuration except for five firmware samples in which we reproduce part of µEMU's configurations. In four of these cases, we disable interrupts (3D_printer, RF_door_lock, Thermostat, xml_parser), while in the fifth case, we add support for DMA operations by manually defining pass-through models for two DMA buffer address MMIO registers (utasker_Mobus). In contrast to FUZZWARE's minimal configuration overhead, the most recent target configurations of µEMU available at the time of writing[3] (1) specify custom definitions of input peripherals, (2) apply custom configurations to tweak exploration parameters to the target, and (3) specify custom path validity information (alive points and/or kill points). This customization requires a human analyst with domain knowledge of the respective target. As noted by the µEMU authors, without further human assistance, P2IM is unable to analyze the 11 samples introduced by µEMU [57].

Hence, we compare the fuzzing performance of all three systems on the 10 targets supported by P2IM, visualized in Figure 5, and provide the data for all experiments, including the ones for the 11 remaining firmware samples in Table 5 in the appendix. The results show that FUZZWARE consistently discovers (significantly) more basic blocks compared to the state of the art. In one case (CNC), FUZZWARE doubles P2IM's coverage and triples µEMU's coverage. For the targets in Figure 5, FUZZWARE yields on average ~44% more code coverage than P2IM and ~61% more coverage than µEMU (~57% when averaged over all targets). In 19 out of 21 times, the minimum coverage achieved by FUZZWARE exceeds the maximum coverage of the prior approaches. In other words, even the worst run of FUZZWARE performs better than the best run of µEMU and P2IM. With the exception of PLC and LiteOS_IOT, where FUZZWARE showed variable performance, all results are statistically significant according to the Mann–Whitney U test, as recommended by Klees et al. [29].

One interesting aspect in Figure 5 is that P2IM often outperforms µEMU. We believe the reason for this to be the aggressive path elimination that is at the core of µEMU's invalidity-guided approach (cf. Section 3): The framework deploys heuristics to decide on viable paths and provides hardware values accordingly. When no clear distinction can be made, this choice is left to randomness, which either makes large parts of the firmware available for analysis or removes them entirely from the ongoing run. This may also explain why individual runs perform better than P2IM. Further, we assume both approaches to path elimination are also responsible for the fact that basic block discovery graphs flatline early. FUZZWARE does not eliminate paths, resulting in a higher code coverage. We discuss path elimination and code coverage in Section 7.

Alongside the significant increase in coverage and automation, FUZZWARE uncovered previously unreported bugs in three targets. Manual root cause analysis showed that FUZZWARE identified one concurrency issue (Soldering Iron), a missing pointer verification (CNC), and an unchecked AT command parsing crash (GPSTracker). For all three targets, the discovery of additional bugs found by FUZZWARE coincides with a significant increase in code coverage.

Regarding RQ 4, the results indicate that our access modeling allows a fuzzer to clearly outperform the current state of the art. Towards RQ 5, we observe that FUZZWARE is able to identify bugs in real-world firmware and also find new bugs that previous work does not locate.

6.3 Fuzzing New Targets

Fuzzing Targets. To expand on RQ 3 and RQ 5, we used FUZZWARE to test different functionalities of the core net-

[3] Our evaluation bases on git commits 5b12949325 and 67e50000bb of the uEmu-real_world_firmware and uEmu repositories, respectively.

work stacks of two popular embedded rmware frameworks: Zephyr [55], and Contiki-NG [12]. Both projects are well-maintained, with hundreds of contributors, and backing companies such as Intel and Google.

We chose the radio layer implementations of these two systems as a fuzzing target. Connected devices heavily rely on network stacks, and the corresponding low-level parsing code exposes a universal attack surface. As successful attacks can potentially propagate from one device to another, aws in these types of interfaces put whole eets of devices at risk.

We based all of the rmware images on code samples demonstrating uses of different network stacks. Similar to the rationale for re-using identical application-level code in Section 6.1, logic within higher layers (such as the application layer) does not in uence the inner workings of lower layers (e. g., low-level radio packet processing).

Bug Case Studies In total, FUZZWARE discovered 12 distinct bugs in these targets, for which 12 CVEs have been assigned after a coordinated disclosure. In the following, we provide case studies for some of these bugs. A full overview of these bugs can be found in Table 6 in the appendix.

CVE-2020-12141 In the tested version of Contiki-NG, the Simple Network Management Protocol (SNMP) parsing logic of incoming SNMP messages did not correctly validate the user-supplied size of the variably-sized community eld. This lead the logic to access the user-supplied buffer out of bounds, resulting in a rmware crash (DoS).

CVE-2021-3321 As a translation layer from radio frames to IPv6 packets, the IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) standard de nes a custom header compression mechanism. Before decompression, Zephyr checked the required size of the decompressed header payload, and would correctly allocate an appropriately-sized destination buffer to hold the decompressed contents. The logic did not check, however, whether the source frame was actually large enough to hold the compressed header payload. As a result, it consumed more bytes from the frame-holding buffer than available, leading to a size eld integer under ow, followed by a corruption of memory.

CVE-2021-3330 To transport IPv6 packets from small radio frames, 6LoWPAN de nes a fragmentation layer. To differentiate between the start and subsequent entries of a list of fragments, frames are assigned the fragment types FRAG1 and FRAGN, respectively. When encountering a FRAGN fragment, the reassembly logic would insert the fragment to the start of the fragment list, and correctly check that its contents are marked for insertion at the beginning of the reassembled buffer. Before reassembling, however, the logic did not check whether a FRAG1 fragment is present. Assuming a FRAG1 fragment to be present, the fragment sorting logic would predicate its algorithm on a pre-sorted rst element. Using a crafted set of input fragments which exactly match the required overall size, but does not contain a FRAG1 fragment, the sorting logic can be tricked into creating an unintended cyclic reference within the list, which translates into an eventual integer under ow, followed by a buffer over ow.

Table 3: Root cause categories of unique crashes generated by FUZZWARE.

Firmware Set	#Unique Crashes	#Security Issues	#Unchecked Initialization	#False Positives
Synthetic Samples	10	10	-	-
P2IM	16	9	7	-
µEMU	19	9	9	1
Zephyr	12	10	-	2
Contiki-NG	4	4	-	-
Total	61	42	16	3

This experiment shows that our modeling approach allows a fuzzer to effectively test and nd bugs in well-maintained, widely-used real-world rmware code (RQ 5).

6.4 False Positive Crash Analysis

Finally, we investigated the crashes produced by FUZZWARE. To this end, we deduplicated the crashing test cases generated across the previous experiments and performed a manual root cause analysis. Table 3 shows the results of the experiment.

Our analysis showed that 42 out of the 61 unique crashes corresponded to security issues, and 16 crashes occurred as rmware logic does not robustly handle initialization, e. g., by not checking the return value of initialization APIs. The three remaining test cases related to omitted rmware checks. We identify these three crashes as false positives, since the bugs exist in the rmware, but will not occur on real hardware. Two of these crashes occurred in Zephyr: In the rst case, the length of a radio packet was implicitly assumed to have a maximum value of 127, while a full byte of hardware-generated MMIO value was used without checks in a size variable (maximum value: 255). This leads to a buffer over-ow for size values greater than 127. In the other case, an interrupt handler used a pointer variable without initialization checks. It assumed the variable to be initialized when an interrupt was raised. If an interrupt was raised by the fuzzer before this initialization was performed, interrupt handling would result in a NULL pointer dereference. The third false positive crash was caused in the µEMU utasker_USB sample, where the USB receive channel number register eld CHNUM may have a maximum value of 15, but only less USB channels are actually in use, leading to another out-of-bounds access.

In essence, the results show that we abstract away hardware from rmware in the fuzzing process. This implies that if a bug exists in the software regardless of the hardware environment, FUZZWARE might identify it. However, it does not guarantee that the bug can be triggered in a speci c real hardware deployment. On the contrary, it might just demonstrate that the software developers are trusting a speci c hardware. Becoming aware of these types of issues may have some upside: the same rmware running in a different hardware en-

vironment might suffer from a security vulnerability (since the hardware cannot be trusted). This way, FUZZWARE pinpoints possible security issues, even before the code is deployed in a different hardware environment.

7 Discussion

In this section, we further discuss our design decisions, the applicability of FUZZWARE outside our prototype implementation, and possible future research directions.

Direct Memory Access (DMA). As discussed in Section 4.5, while FUZZWARE allows handling DMA via additional configuration, automatically modeling DMA is not the focus of this work. However, we see one central contribution of this work towards automated DMA handling: As DMA-handling firmware code is often part of more complex code (where, in practice, heavy MMIO use is inevitable), achieving a high baseline code coverage (as we show is the case with FUZZWARE in Section 6) is a prerequisite to even triggering any use of DMA. The authors of DICE [37], while recently describing a generic DMA handling approach, encountered this issue: Their evaluation shows that previous firmware fuzzing systems are unable to reach DMA logic for more than a third (36%) of detected DMA, which can likely be attributed to missing firmware code coverage.

Using Access Models outside FUZZWARE. The tight integration of FUZZWARE's MMIO access models in the fuzzing process raises the question whether those models can be used independently. Luckily, once these models are generated, the only information needed to serve a request are access size, location, and program counter value – information which is readily available in other analysis frameworks. To demonstrate that this allows interoperability, we integrated FUZZWARE generated models into avatar^2 [38] as so-called hyperipherals. This enables dynamic analysis capabilities beyond fuzzing, such as taint analysis using the PANDA framework [16].

Merits of Path Elimination. Previous approaches (e.g., [7, 18, 57]) eliminate code paths to steer firmware execution. As we describe in Section 3.2, eliminating paths bears the risk of excluding relevant functionality from the analysis, either by removing error handling or by forcing execution into complex error handling routines, away from ordinary functionality. As a heuristics-based classification of "correct" code paths is error prone, and a misclassification requires manual intervention to remediate, we aimed for robustness in a fully automated setting by avoiding such classification attempts. To facilitate automation, FUZZWARE allows hitting stuck cases (such as hitting tight infinite error loops), and relies on the fuzzing engine to avoid them based on timeouts and coverage feedback. Future work could improve upon this in a middle ground approach by identifying and eliminating stuck cases that can be safely removed without reducing the amount of reachable firmware logic.

Implicit Peripheral Semantics. During our evaluation, we found that implicit assumptions made by firmware about the behavior of its surrounding hardware account for some of the crashing test cases. The underlying notion could open up potential future research: Intuitively, we assume that the way in which firmware code is built and operates exposes information about the implicit assumptions it makes about its surrounding hardware. This may include assumed size limits, as well as the expected order in which certain events are assumed to occur. Further analyses could use this type of information to derive increasingly complex models of peripheral behavior.

8 Related Work

Coverage-guided fuzzers, such as AFL [54] and more advanced approaches [3, 6, 19, 41, 47, 52], have found numerous critical bugs in major applications and OSes. One important line of research focuses on increasing the quality of fuzzing inputs, for instance via taint tracking [10, 41], symbolic execution [20, 47, 52], or by additional program state analysis [2, 3]. While we can leverage the described techniques to improve our work, the fuzzers implementing them target desktop applications and are not directly applicable to embedded systems.

Multiple studies apply black-box fuzzing to embedded devices [9, 31, 40], but generally suffer from a lack of coverage information and the inability to reset the device to a known state. This challenge is commonly overcome by re-hosting embedded systems' firmware [17, 51]. Recent work uses QEMU [5], especially for emulating Linux-based firmware [8, 14, 28, 56]. Unfortunately, these approaches heavily rely on the abstractions provided by the Linux kernel and, thus, are not applicable to monolithic firmware as analyzed in our work.

Similar to using abstractions provided by the Linux kernel, one rehosting approach builds on top of the hardware abstraction layers present in many firmware images [11, 33, 35]. Unfortunately, identifying and modeling those abstractions still requires target-specific knowledge and manual effort, even with the automation presented in HALUCINATOR [11].

Hardware-in-the-loop approaches (e. g., [13, 26, 27, 30, 38, 48, 53]) avoid the need for abstractions by forwarding hardware accesses to a physical device during emulation. While this allows for dynamic analysis of firmware, these approaches have limited applicability to fuzz testing. On top of forwarding being a typical bottle-neck for most of these systems, they require one instance of the target hardware per fuzzing thread, as the hardware state and fuzzer must be kept consistent.

The recent trend of pattern-based MMIO modeling was introduced by PRETENDER [22], which still required a hardware-in-the-loop recording phase. This hardware dependency was later resolved by P2IM [18], and then addressed by μEMU [57], as extensively discussed in Sections 3 and 6.

Various further approaches integrating dynamic symbolic execution in the rehosting process to infer correct values for hardware accesses have been proposed [7, 15, 24, 34].

LAELAPS [7], a recent and representative approach targeting monolithic firmware, allows to steer the execution to interesting locations by involving a human analyst. Unlike FUZZWARE, this human-in-the-loop approach is not designed for automated fuzz testing, as fuzzing a specific firmware image with LAELAPS requires significant target-specific manual effort, and its emulation does not scale well due to frequent invocations of the expensive symbolic execution engine.

9 Conclusion

In this work, we presented a novel approach for modeling MMIO interactions to effectively fuzz test a monolithic firmware binary. Our access models are based on deeper insights into firmware logic and, consequently, allow one to eliminate types of input overhead that have previously been inaccessible to existing MMIO modeling approaches. Applying these models results in a drastically improved fuzzing effectiveness over the current state of the art.

10 Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by NWO under 628.001.030 "Tropics" and NWA-ORC Inter-Sect. In addition, this material is based upon work supported by DARPA under agreement number HR001118C0060, and by ONR under agreements N00014-17-1-2011 and N00014-17-1-2897. This material is based on research sponsored by The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the of cial policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] ARM. DUI 0552A: Cortex-M3 devices generic user guide, 2019.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In IEEE Symposium on Security and Privacy, 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In Symposium on Network and Distributed System Security (NDSS), 2019.

[4] Aspencore. Embedded markets study: Integrating IoT and advanced technology designs, application development & processing environments. EETimes Embedded, 2019.

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

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In ACM Conference on Computer and Communications Security (CCS), 2016.

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

[8] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In Symposium on Network and Distributed System Security (NDSS), 2016.

[9] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In Symposium on Network and Distributed System Security (NDSS), 2018.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In IEEE Symposium on Security and Privacy, 2018.

[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, 2020.

[12] Contiki-NG. https://github.com/contiki-ng/contiki-ng, 2020. Accessed: October 5, 2021.

[13] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In USENIX Security Symposium, 2018.

[14] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2016.

[15] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on rmware: Finding vulnerabilities in embedded systems using symbolic execution. In USENIX Security Symposium, 2013.

[16] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In Program Protection and Reverse Engineering Workshop, 2015.

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

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

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

[20] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In Symposium on Network and Distributed System Security (NDSS), 2008.

[21] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. AntiFuzz: Impeding fuzzing audits of binary executables. In USENIX Security Symposium, 2019.

[22] 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 rmware through automated re-hosting. In Symposium on Recent Advances in Intrusion Detection (RAID), 2019.

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

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

[25] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzi cation: Anti-fuzzing techniques. In USENIX Security Symposium, 2019.

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

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

[28] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot rmware for dynamic analysis. In Annual Computer Security Applications Conference (ACSAC), 2020.

[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In ACM Conference on Computer and Communications Security (CCS), 2018.

[30] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In USENIX Workshop on Offensive Technologies (WOOT), 2015.

[31] Karl Koscher, Stefan Savage, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Alexei Czeskis, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In IEEE Symposium on Security and Privacy, 2010.

[32] Edward Ashford Lee and Sanjit Arunkumar Seshia. Introduction to Embedded Systems: A Cyber-Physical Systems Approach. The MIT Press, 2nd edition, 2016.

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

[34] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. Mousse: a system for selective symbolic execution of programs with untamed environments. In European Conference on Computer Systems, 2020.

[35] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE. In ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec), 2020.

[36] Mbed OS. https://www.mbed.com/en/platform/mbed-os/, 2020. Accessed: October 5, 2021.

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

[38] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In Workshop on Binary Analysis Research (BAR), 2018.

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

[40] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of death: From analyzing to attacking mobile phones on a large scale. In USENIX Security Symposium, 2011.

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In Symposium on Network and Distributed System Security (NDSS), 2017.

[42] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-Cube: High-dimensional hypervisor fuzzing. In Symposium on Network and Distributed System Security (NDSS), 2020.

[43] Edward Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In IEEE Symposium on Security and Privacy, 2010.

[44] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In Symposium on Network and Distributed System Security (NDSS), 2015.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK: (state of) the art of war: Offensive techniques in binary analysis. In IEEE Symposium on Security and Privacy, 2016.

[46] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: automated iot firmware introspection and analysis. In ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P), 2019.

[47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In Symposium on Network and Distributed System Security (NDSS), 2016.

[48] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In USENIX Security Symposium, 2018.

[49] Unicorn Engine. https://www.unicorn-engine.org/, 2017. Accessed: October 5, 2021.

[50] Wind River SIMICS. https://www.windriver.com/products/simics/, 2020. Accessed: October 5, 2021.

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

[52] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In USENIX Security Symposium, 2018.

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

[54] Michal Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017. Accessed: October 5, 2021.

[55] Zephyr Project. https://www.zephyrproject.org/, 2020. Accessed: October 5, 2021.

[56] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In USENIX Security Symposium, 2019.

[57] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, 2021.

A Appendix

Table 4: Hardware platforms and firmware samples used in FUZZWARE's evaluation.

Platform	Firmware Samples
ARCH_PRO	Password_Discovery
EFM32GG_STK3700	Password_Discovery
EFM32LG_STK3600	Password_Discovery
LPC1549	Password_Discovery
LPC1768	Password_Discovery
MAX32600	RF_Door_Lock, Thermostat
MK64FN1M0VLL12	P2IM unit tests, Console
MOTE_L152RC	Password_Discovery
NUCLEO_F207ZG	Password_Discovery
SAM3X8E	P2IM unit tests, Heat_Press, Steering_Control
SAM3X/A	GPS tracker
SAMR21	6LoWPAN_Sender, 6LoWPAN_Receiver
STM32F103RB	Password_Discovery, P2IM unit tests, Drone, Gateway, Reflow_Oven, Robot, Soldering_Iron
STM32F103RE	3Dprinter
STM32F429ZI	CNC, PLC, utasker_MODBUS, utasker_USB
STM32L152RE	Password_Discovery, XML_Parser
STM32L431	LiteOS_IoT
STM32L432KC	Zepyhr_SocketCan
UBLOX_C027	Password_Discovery

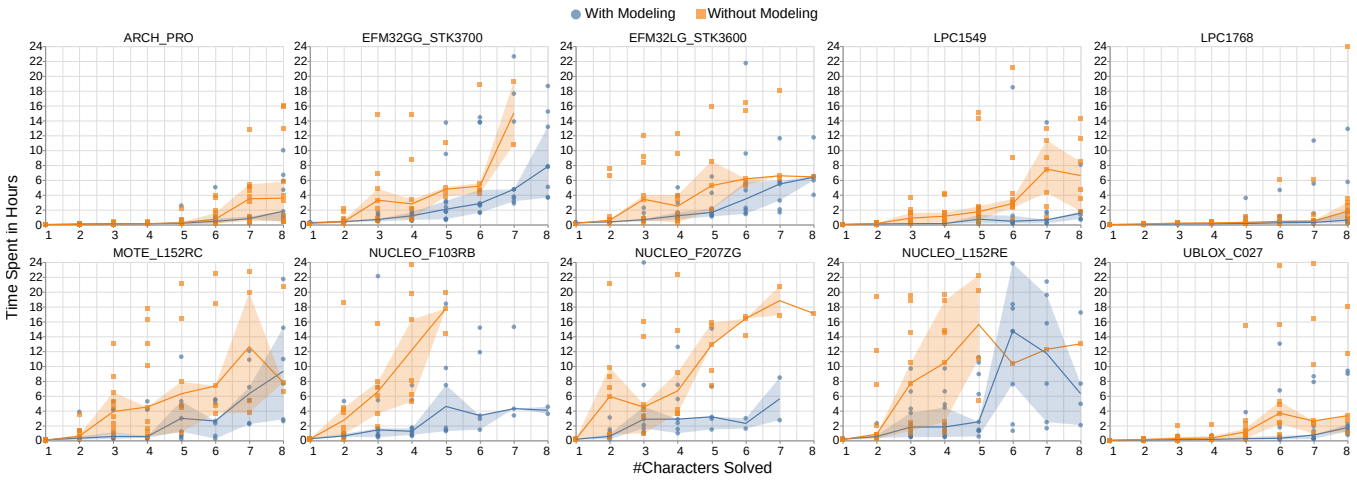

Figure 6: Time spent by FUZZWARE for character discovery on 10 synthetic firmware samples over ten 24h runs with and without modeling. Shown are individual timings (dots), the mean and 66% intervals. Each dot represents the point of time at which the character was solved by one run. Thus, if all ten runs succeed in finding a character, ten dots exist for this character. A high number of dots indicates consistency in finding the character, while dots positioned low indicate high speed in solving that character.

Table 5: FUZZWARE (FW.) vs P2IM vs μEMU coverage generation over five 24-hour fuzzing iterations. We compare the minimum (#BB min), average (#BB avg), maximum (#BB max), and total number of basic blocks (#BB total) discovered across all runs by each system. We account for targets used by both P2IM and μEMU (upper part) and such only evaluated by μEMU (lower part). Bold numbers indicate the best result in each category. The last two columns show the p-value according to the Mann-Whitney U test between the runs of FUZZWARE and P2IM, and between FUZZWARE and μEMU. For all but two cases the p-value ($p < 0.01$) indicates statistical significance of the results.

Target	**#BB in**	**#BB min**			**#BB avg**			**#BB max**			**#BB total**			p-value	
	target	P2IM	μEMU	FW.	P2IM	μEMU	FW.	P2IM	μEMU	FW.	P2IM	μEMU	FW.	to P2IM	to μEMU
CNC	3614	1096	416	**2422**	1252	786	**2560**	1578	1136	**2646**	1599	1140	**2722**	< 0.01	< 0.01
Drone	2728	1268	1456	**1830**	1270	1457	**1836**	1275	1459	**1847**	1275	1461	**1850**	< 0.01	< 0.01
Heat Press	1837	527	492	**537**	532	493	**544**	536	493	**547**	536	494	**550**	< 0.01	< 0.01
Reflow O.	2947	815	797	**1188**	815	829	**1189**	815	875	**1191**	815	880	**1191**	< 0.01	< 0.01
Soldering I.	3656	1302	837	**2080**	1302	947	**2117**	1302	1271	**2134**	1302	1283	**2145**	< 0.01	< 0.01
Console	2251	779	583	**805**	779	615	**805**	779	662	**805**	779	662	**805**	< 0.01	< 0.01
Gateway	4921	1756	1623	**2423**	1768	1738	**2622**	1804	1905	**2881**	1806	1977	**2984**	< 0.01	< 0.01
PLC	2303	**505**	436	465	507	436	**603**	513	436	**647**	513	451	**649**	0.07	< 0.01
Robot	3034	1131	999	**1267**	1158	1004	**1296**	1190	1014	**1340**	1192	1017	**1340**	< 0.01	< 0.01
Steering C.	1835	498	489	**598**	498	497	**609**	498	506	**613**	498	506	**613**	< 0.01	< 0.01
6LoWPAN_Recv.	6977	-	2477	**3056**	-	2501	**3099**	-	2520	**3142**	-	2572	**3155**	-	< 0.01
6LoWPAN_Send.	6980	-	1688	**2914**	-	2342	**3066**	-	2522	**3144**	-	2550	**3166**	-	< 0.01
RF_door_lock	3320	-	605	**782**	-	664	**1675**	-	679	**2262**	-	679	**2641**	-	< 0.01
Thermostat	4673	-	907	**2274**	-	936	**2747**	-	980	**3082**	-	1020	**3545**	-	< 0.01
3D_printer	8045	-	854	**889**	-	854	**977**	-	854	**1217**	-	855	**1221**	-	< 0.01
GPSTracker	4194	-	587	**1006**	-	588	**1016**	-	588	**1027**	-	602	**1040**	-	< 0.01
LiteOS_IOT	2423	-	657	**737**	-	740	**954**	-	804	**1342**	-	804	**1343**	-	0.26
utasker_Modbus	3780	-	1043	**1247**	-	1049	**1297**	-	1057	**1326**	-	1113	**1327**	-	< 0.01
utasker_USB	3491	-	594	**1587**	-	961	**1669**	-	1121	**1718**	-	1150	**1807**	-	< 0.01
zephyr_socket.	5943	-	2176	**2553**	-	2282	**2722**	-	2396	**2869**	-	2456	**2884**	-	< 0.01
xml_parser	9376	-	1717	**3185**	-	1781	**3602**	-	1861	**4012**	-	1955	**4334**	-	< 0.01

Table 6: Overview of CVE-Assigned Bugs found by FUZZWARE

CVE	**Product**	**Version Tested**	**Description**
CVE-2020-10064	Zephyr OS	2.2.0	Improper Input Frame Validation in ieee802154 Processing
CVE-2020-10066	Zephyr OS	2.2.0	Incorrect Error Handling in Bluetoot HCI core
CVE-2020-10065	Zephyr OS	2.2.0	Missing Size Checks in Bluetooth HCI over SPI
CVE-2020-12141	Contiki NG	4.4	Missing size check during SNMP message decoding
CVE-2020-12140	Contiki NG	4.4	Details omitted due to patch status
CVE-2021-3319	Zephyr OS	@d969ac..1cc42d	Incorrect 802154 Frame Validation for Omitteed Source/Dest Address
CVE-2021-3320	Zephyr OS	@d969ac..1cc42d	Type Confusion in 802154 ACK Frames Handling
CVE-2021-3321	Zephyr OS	@d969ac..1cc42d	Integer Underflow in IEEE 802154 Fragment Reassembly Header Removal
CVE-2021-3322	Zephyr OS	@d969ac..1cc42d	Unexpected Pointer Aliasing in IEEE 802154 Fragment Assembly
CVE-2021-3323	Zephyr OS	@d969ac..1cc42d	Integer Underflow in 6LoWPAN IPHC Header Uncompression
CVE-2021-3329	Zephyr OS	@d969ac..1cc42d	Details omitted due to patch status
CVE-2021-3330	Zephyr OS	@d969ac..1cc42d	Linked-list corruption leading to large out-of-bounds write while sorting for forged fragment list