AidFuzzer: Adaptive Interrupt-Driven Firmware Fuzzing via Run-Time State Recognition

Jianqiang Wang[*], *Qinying Wang*[†], *Tobias Scharnowski*[*], *Li Shi*[§], *Simon Woerner*[*], *Thorsten Holz*[*]
[*]*CISPA Helmholtz Center for Information Security,* [†]*Zhejiang University,* [§]*ETH Zurich*
Email:{jianqiang.wang, tobias.scharnowski, simon.woerner, holz}@cispa.de
wangqinying@zju.edu.cn, lishil@student.ethz.ch

Abstract

Fuzzing has proven to be an effective method for discovering vulnerabilities in firmware images. However, several hard-to-bypass obstacles still block the way for fuzzers to achieve higher code coverage in the firmware fuzzing process. One major issue is *interrupt handling*, which is fundamental to emulate the firmware: If interrupts are triggered incorrectly, the firmware may crash or get stuck, even at an early stage. Thus, a proper mechanism for triggering and handling interrupts is a crucial yet under-researched aspect of firmware fuzzing.

In this paper, we present AidFuzzer, an adaptive interrupt-driven firmware fuzzing method, to tackle the interrupt triggering problem. The key observation is that firmware images commonly exhibit a consistent *run-time state transition cycle*. In each state, the firmware may require specific interrupts to continue running, or it may not need any interrupts to continue processing data. Based on this observation, we model the type and status of the interrupts to verify that they are exactly the interrupts that the firmware needs at a specific point in time. Moreover, we monitor the run-time state of the firmware and trigger certain interrupts when the firmware expects them or let the firmware run when it does not require interrupts. We have implemented a prototype of AidFuzzer and evaluated it on 10 open-source firmware projects, including well-known real-time operating systems such as RT-Thread and Apache Mynewt-OS. The experiment demonstrates that our framework outperforms state-of-the-art works in terms of coverage when dealing with complex interrupt handling. We also discovered eight previously unknown vulnerabilities in the tested firmware images.

1 Introduction

Over the past decade, embedded devices such as factory robots, medical devices, satellites, and smart fitness bands have become widespread. To streamline the development of firmware for these devices, real-time operating systems (RTOS) are commonly used. Despite advances in firmware development, security threats and software faults still persist. Developers often prefer memory-unsafe languages such as C and C++ for low-level hardware manipulation, but these languages also introduce memory corruption vulnerabilities due to their inherent direct memory access features.

Fuzz testing (*fuzzing*) has proven to be an effective and efficient method for uncovering firmware vulnerabilities. Embedded devices with low performance and slow speed are not inherently designed for fuzzing. Therefore, running firmware in an emulated environment and simulating its peripherals' behaviors—a technique known as *re-hosting*—is a promising approach to improve testing. Tools like P2IM [1] and µEmu [2] attempt to model peripheral behavior by extracting information from the MCU documentation or using symbolic execution. Unfortunately, these techniques are unstable and imprecise. The Fuzzware framework [3] models the MMIO (memory-mapped I/O, the way that the firmware communicates with the peripherals) access into several categories, such as *bitextract*, *passthrough*, and *constant-value*. This MMIO modeling efficiently reduces the input overhead. For instance, the constant-value model only accepts a single specific input (i.e., the input overhead is 1), reducing fuzzer effort on mutating that MMIO access. Hoedur [4], another advanced firmware fuzzing framework, divides a single fuzzing input into multiple streams based on the MMIO access context. This so-called *multi-stream fuzzing* prevents the "avalanche effect", where a value meant for one MMIO access is mistakenly consumed by another. Recently, SafireFuzz [5] proposed binary rewriting of ARM Cortex-M firmware to make it compatible with high-performance ARM Cortex-A processors, aiming to accelerate fuzzing speed. While Fuzzware and Hoedur solely focus on the MMIO input, SafireFuzz requires the presence of a hardware abstraction layer (HAL) to inject the fuzz input into both MMIO and DMA (Direct Memory Access).

Despite advances in firmware fuzzing, including state-of-the-art frameworks, the challenge of accurately triggering interrupts remains unresolved. P2IM models all peripheral behaviors, including interrupts, by analyzing the documentation, but this approach lacks precision. Both Fuzzware and

Hoedur use a round-robin mechanism for interrupt triggering by default, which involves activating an interrupt at regular, fixed time intervals. This process starts with the first enabled interrupt, triggers it, and then triggers the next enabled one. Once all enabled interrupts have been triggered, the cycle repeats with the first enabled one. Fuzzware and Hoedur also support an advanced fuzz-mode interrupt triggering mechanism, which triggers interrupts depending on the fuzzing input. SafireFuzz uses indirect call-level counters and manual clock-update hooks. The design is similar to the round-robin mechanism for timer interrupts from a high-level perspective. While these mechanisms are effective for firmware with straightforward interrupt-triggering conditions, they fall short in more complex scenarios.

Unfortunately, interrupt processing in real-world firmware is often complicated in practice. For example, certain interrupts may be enabled but not yet ready to be triggered because the associated data, such as pointers, are not fully initialized. The round-robin and fuzz-mode interrupt triggering mechanisms do not take the status of the interrupt into account. Therefore, if the fuzzer triggers an interrupt *before* the data is initialized, this can lead to an unexpected crash that brings the fuzzing process to a halt. Even after all data is initialized, triggering some interrupts can cause the firmware to reset or get stuck in an infinite loop. These interrupts should never be triggered, as they hinder the fuzzing progress. In addition, the round-robin and fuzz-mode mechanism triggers interrupts at regular intervals, but this approach has disadvantages: If the interval is too small, the fuzzer will constantly interrupt the execution of the firmware, while too large an interval will cause the firmware to wait for the interrupt. To summarize, we need to answer three questions when dealing with the interrupt-triggering problem:

1. *When* should the interrupts be triggered?
2. *How often* should the interrupts be triggered?
3. *Which* interrupts should be triggered?

We have found that the key observation to answer the above questions is the *run-time state transition* of the firmware. The firmware goes through an initialization phase at boot time and then transitions to a *processing state* where it processes inputs. Once the processing of the inputs is complete, it enters a *waiting state* in which it awaits certain asynchronous events or new inputs, which are usually delivered via interrupts. The cycle then repeats itself when it returns to the processing state. The interrupts should only be triggered when the firmware is in the waiting state, without intervening during the processing state. By automatically analyzing the *interrupt service routine* (ISR), we can pinpoint which interrupts are able to transition the firmware from the waiting state to the processing state. We refer to these as *effective interrupts*. When the firmware is in a waiting state, we selectively trigger only the effective interrupts that can cause a transition to the processing state. We also make sure that each interrupt we want to trigger is ready to be triggered, e.g., by checking the initialization of the associated data. AIM [6] proposes a similar interrupt analysis method for firmware testing. It is based on the same idea that ISRs can influence the behavior of the firmware. However, AIM does not model the firmware run-time state and cannot reveal the relationship between the run-time state and the interrupt. Therefore, it cannot answer the three questions accurately. In addition, the overall design and implementation of AIM is based on symbolic execution, which significantly affects the analysis speed.

In this paper, we introduce AidFuzzer, an ***A***daptive ***I***nterrupt-***D***riven Fuzzing framework that provides a proper interrupt triggering mechanism for firmware fuzzing. AidFuzzer identifies effective interrupts, triggers interrupts on demand, and only triggers the interrupts that are required by the firmware. To evaluate the performance of AidFuzzer, we compiled a collection of 10 open-source firmware projects based on the ARM Cortex-M processor. This dataset includes open-source Github projects and popular RTOS examples, such as RT-Thread and Apache Mynewt-OS. Our experimental results show that AidFuzzer performs better than the state-of-the-art tools Fuzzware, Hoedur, and SafireFuzz in handling complex interrupt scenarios. In addition, we found eight previously unknown vulnerabilities in these open-source projects.

Contributions We make the following key contributions:

- We are the first to systematically discuss the interrupt triggering problems in firmware fuzzing and propose the use of an adaptive interrupt triggering mechanism to overcome the complex interrupt situations encountered in real-world firmware fuzzing.
- We identify the key insight to solve the interrupt triggering problem: the run-time state transition cycle of a running firmware and the relations between the interrupt triggering and run-time state. Based on this insight, we implement an adaptive interrupt-driven firmware fuzzing prototype called AidFuzzer.
- We tested AidFuzzer on 10 open-source firmware projects. AidFuzzer outperforms existing work in handling complex interrupt scenarios, and we found eight previously unknown vulnerabilities.

To foster future research on firmware fuzzing, we are making the prototype of AidFuzzer available as open source at https://github.com/wjqsec/aidfuzzer.

2 Technical Background

Since we evaluate AidFuzzer on ARM Cortex-M based firmware targets, and we focus on the interrupt handling problem, we now provide a brief introduction to the ARM Cortex-M nested vector interrupt control (*NVIC*).

2.1 IRQ and Interrupt Vector Table

An interrupt request (*IRQ*) is an asynchronous event typically initiated by peripherals. Exception handling by the processor

follows a similar path to that of an interrupt, but exceptions are internally generated by the processor, such as encountering an illegal instruction or a division-by-zero fault. This paper specifically focuses on asynchronous interrupts originating from peripherals. We will use IRQ triggering and interrupt triggering interchangeably in the paper, as they both refer to the same concept: A device sends an asynchronous interrupt request to the processor.

When an interrupt occurs, it alters the control flow from the current processor execution context (referred to as *thread mode* in ARM Cortex-M) to the interrupt handling context (referred to as *handler mode*). Upon detecting the incoming interrupt signal, the processor automatically preserves the current register context by saving it to the stack. Subsequently, it retrieves the IRQ number and the corresponding Interrupt Service Routine (*ISR*) address from the *interrupt vector table* and starts executing the ISR.

The interrupt vector table is an array of function pointers in memory that use the IRQ number as an index to access its elements. For instance, consider an interrupt vector table located at address 0x20000000 and an IRQ with a number of 0x20, the processor finds the corresponding ISR at address 0x2000000080 (calculated as 0x20000000 + 4 * 0x20, as a function pointer in the ARM Cortex-M is 4 bytes in size). The processor then loads the memory content from 0x2000000080 into the Program Counter (PC). Upon completing the ISR, the processor loads the *EXC_RETURN* [7] value, previously saved in the link register during the context switch, into the PC, indicating an interrupt exit. The processor automatically restores the previous context and resumes execution from thread mode.

Each interrupt has a corresponding priority, and when multiple interrupts occur simultaneously, the processor gives priority to the one with the highest priority. It is noteworthy that this paper excludes the handling of nested interrupts and tail interrupts, as interrupts are only triggered in thread mode, and only one interrupt is triggered at a time in our implementation.

Note that the interrupt vector table is subject to dynamic re-basing, and its elements can be overwritten during runtime. For instance, the firmware might alter the table address from 0x20000000 to 0x30000000 or overwrite the memory content at address 0x2000000080. Consequently, different ISRs can be employed in such scenarios to handle the same IRQ. This flexibility in re-configuring the table allows for dynamic adjustments to the interrupt handling process, enabling the system to adapt to changing requirements or respond to specific run-time conditions.

2.2 NVIC Configuration

NVIC is mapped as a Memory-Mapped I/O (*MMIO*) region and can be configured by writing to this designated memory space. As an example, the vector table base address can be configured by the firmware. When the firmware requires NVIC configuration, it simply writes to the pertinent field within the NVIC data structure. In this paper, we focus on the following NVIC configurations:

- *Enable/Disable IRQ*: An array of bits indicates the enable/disable status of an IRQ. NVIC supports up to 240 interrupts [7]. However, besides the reserved interrupts that are enabled by default, only a number of the IRQs are used and enabled by the firmware. The IRQ can only be used by the peripherals and triggered if it is enabled.
- *Interrupt Vector Table Base*: Writing to this field changes the table base address. The processor fetches the subsequent ISR address based on the updated value.
- *IRQ Pending*: An array of bits indicates pending IRQ requests. Writing to this field pends a corresponding IRQ, waiting to be handled by the processor. Note that setting a pending bit to this array does not mean that this IRQ will be served immediately, it also depends on several other conditions: a) If interrupt handling is enabled globally by the processor. b) If the specific IRQ is enabled in the NVIC configuration.

One bit in the Current Program Status Register (*CPSR*) for ARM processors indicates the global interrupt enable/disable status. Setting/clearing this bit allows/prevents all the interrupts from being served.

2.3 Types of NVIC Interrupts

IRQ numbers 1-15 are usually reserved for core ARM Cortex-M processor functionalities such as reset, system-call, and hard fault, which are not triggered by peripherals. One exception is the SysTick IRQ: Firmware may rely on it to accomplish its functionalities such as task scheduling. Peripherals customize other IRQs mainly to handle the following situations: a) New data is available either from DMA buffers or from device registers. b) Output data is consumed or processed by the peripherals. c) A specific time interval has passed. d) There is an internal status change in the peripherals. e) Other unexpected event happens. For example, a typical 8250 UART device [8] ISR either reads a character from the Receiver Buffer register when a new input character arrives or writes a cached character to the Transmitter Holding Buffer register when the device is ready to consume more characters.

2.4 QEMU NVIC Implementation

QEMU maps the NVIC as an MMIO region and maintains the NVIC status in its internal data structure (i.e., this internal structure belongs to QEMU instead of the emulated fimware). When the virtual machine accesses this memory region, a corresponding QEMU function is called. For example, when a virtual machine writes to the enable/disable IRQ bit array, the function *nvic_sysreg_write* serves this operation. The IRQ status will be updated in this function, and the status is persisted in *NVICState* structure. QEMU provides a function

called *armv7m_nvic_set_pending* for the peripherals to trigger an interrupt. In this function, QEMU checks if the IRQ is allowed to be triggered, calculates the priorities among all the pending IRQs, makes the highest priority IRQ active, and notifies the execution thread about the new interrupt request. The execution thread regularly checks if there are pending interrupt requests. If any, it performs the context switch and starts executing the ISR.

3 Motivation Examples

In this section, we use concrete examples to show why triggering interrupts is crucial for firmware fuzzing. All examples are adapted and simplified based on real firmware. In the first example, an infinite loop dummy ISR serves an IRQ during the initial stage to prevent the firmware from running into an unintended state. The second example shows a watchdog ISR that halts the system when an unexpected error happens. These two IRQs should never be triggered since they immediately stop the firmware running, thus hindering the fuzzing process. In the third example, a UART ISR extracts characters from a ring buffer and outputs them to the terminal; it should be triggered only when necessary. By using these three examples, we illustrate what concrete problems should be considered when triggering an interrupt.

3.1 Dummy ISR Example

Listing 1: A commonly used dummy ISR

```
void dummy_isr() {
    while(1) { ; }
}
```

An infinite loop as shown in Listing 1 is commonly used by the firmware to implement a dummy ISR during the initial stage. The firmware usually first enables the IRQ and then initializes the actual ISR later. In practice, there is even firmware that activates certain IRQs and allows them to be served by dummy ISRs during the entire run-time. Moreover, some developers use a function pointer in the ISR as a dummy function and initialize the pointer afterwards. The firmware directly de-references the pointers without NULL checking since the IRQ will only be triggered after the data are properly initialized in a real environment. However, if the interrupt is triggered too early before it is fully initialized, the firmware gets stuck or crashes, preventing the fuzzer from making any progress in fuzzing. The firmware works well in the real environment because no peripherals use this interrupt before they are fully initialized. The round-robin and fuzz-mode interrupt mechanisms will eventually trigger it and hinder the fuzzing process.

3.2 Simplified Watchdog ISR Example

Listing 2: A simplified watchdog ISR

```
void watchdog_isr() {
    trace_watchdog_isr_event();
    if(wdt_handler)
        wdt_handler();
    system_hal();
}
int main() {
    do_something1();
    watchdog_tickle();
    do_something2();
    watchdog_tickle();
}
```

Even after initializing the ISR and the data, some IRQs should never be triggered during fuzzing. Listing 2 shows a watchdog ISR. The firmware needs to regularly tickle the watchdog to prevent it from triggering a watchdog interrupt. In the watchdog ISR, the watchdog interrupt event is logged, the handler provided by the firmware is called to perform the cleaning task, and finally the system is reset. The implementation of *system_hal* depends on the specific board. It may simply go into an infinite loop or execute a breakpoint instruction. The watchdog ISR is used to prevent the firmware from corrupting the data when something unexpected happens. Normally, this interrupt is not expected to be triggered during regular execution, as the watchdog is regularly tickled. However, in a round-robin or fuzz-mode interrupt mechanism, it will be triggered at some time, thus hindering the fuzzing process.

3.3 Simplified UART ISR Example

Listing 3: A simplified UART ISR

```
struct ring_buffer output_buffer;
void uart_tx_isr() {
  if(uart_ready() && !empty(&output_buffer)) {
    char c = dequeue(&output_buffer);
    uart_write(c); // write to uart register
  }
}
void uart_tx_string(char *s) {
  while (*s) {
    while(full(&output_buffer)) {
        ;  // stuck here
    }
    enqueue(&output_buffer,*s);
    s++;
  }
}
void main() {
  uart_tx_string("before do something");
  do_something();
  uart_tx_string("hello world");
}
```

Unlike the watchdog interrupt, Listing 3 shows a simplified UART ISR that the firmware relies on to accomplish its console output functionality which should be triggered when necessary. Listing 3 defines a ring buffer *output_buffer*. The functions *full* and *empty* check whether the buffer is full or empty. When the UART interrupt gets triggered, it checks if the device is ready to consume more characters and, if so, it fetches a character from the ring buffer and writes it to the console. The function *uart_tx_string* is used to output a message string. It keeps looping until all the string characters are inserted into the ring buffer. In the *main* function, the firmware outputs several messages. However, to avoid getting stuck in the *uart_tx_string* function, the UART interrupt must be triggered to consume the characters from the ring buffer. The round-robin or fuzz-mode mechanism can work in this situation. However, a short trigger time interval interferes with the execution of the function *do_something*, while a long interval makes the firmware busy checking the ring buffer in the function *uart_tx_string*.

3.4 Lessons Learned

Recall the three questions we aim to answer in this paper:

1. *When* should the interrupts be triggered?
2. *How often* should the interrupts be triggered?
3. *Which* interrupts should be triggered?

Learning from the above examples, we propose the following heuristics:

1. Interrupts should be triggered only after the ISR and the data are initialized. We call this IRQ status *ready*. (*When*)
2. Interrupts should only be triggered when the firmware needs them. We call this firmware run-time state *waiting*. For example, when the firmware keeps checking the status of the ring buffer in the simplified UART ISR example, it is in a waiting state. (*How often*)
3. Only the interrupts whose ISR can change the firmware run-time state from waiting to not waiting should be triggered. We call this IRQ type *effective*. For example, the ISRs in the first two examples make the firmware get stuck and are thus not effective IRQs. However, the UART ISR in the third example lets the firmware escape from the waiting state and continue running, thus it is an effective IRQ. (*Which*)

In summary, to solve the interrupt triggering problem, we need to identify the IRQ status (ready or unready), IRQ types (effective or ineffective), and the firmware run-time state (waiting or not waiting).

4 Challenges and Insights

Solving the problem of interrupt-triggering is a challenge in practice due to the complexity of the interrupt design. Identifying the IRQ status, IRQ types, and firmware run-time state is a non-trivial task.

4.1 Challenges

The complicated interrupt design in real-world firmware poses two main challenges:

- When the processor serves an IRQ, it retrieves the interrupt vector table base address and indexes the ISR using the IRQ number. The vector table is subject to dynamic re-basing, and its elements can be overwritten at run-time. Thus, a single IRQ number can be served by multiple ISRs. Besides, function pointers are widely used in ISRs. A function pointer can point to different functions at firmware run-time. An ineffective IRQ may become an effective one after any of these conditions gets changed. Therefore, the type and the status of the IRQ cannot be statically determined. We summarize this challenge as *run-time data dependency*.
- Manually analyzing the firmware to determine the run-time state requires a non-trivial amount of work. The firmware does not enter a waiting state at a fixed time interval, but is highly dependent on the program logic. The firmware requires different interrupts in different waiting states. Statically analyzing the entire firmware to determine when the firmware enters a waiting state is a tedious task, as static analysis is not scalable. We summarize this challenge as *state recognition*.

4.2 Insights

To solve the run-time data dependency challenge, we monitor and intercept the changes of the interrupt vector table base, vector table entries, and the function pointers used in the ISR during the whole fuzzing campaign. If there is any change, we extract the ISR address from the vector table, dump the firmware registers and memory, analyze it, and save the analysis results in an IRQ model database. When an update is detected, e.g., when a function pointer is overwritten with a new value, we first try to find the model in our IRQ model database. If we find the corresponding model for the update, we apply it; otherwise, we perform a re-analysis. In this way, we always keep using the latest IRQ model.

For the state recognition problem, we observe that most firmware share a common run-time transition cycle. As shown in Figure 1, the firmware boots itself and then goes into an infinite processing-waiting loop. In the processing state, it does not require any interrupts and is busy processing data. In the waiting state, it requires interrupts to change its state back to processing again. We have the following key observation:

> The effective IRQs change the firmware run-time state from waiting to processing by modifying global objects.

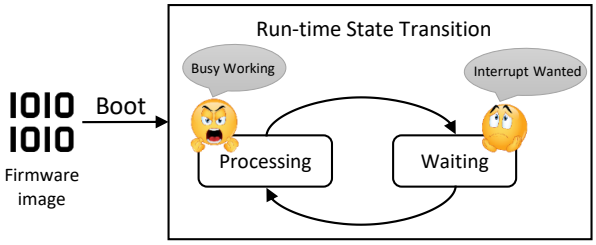

Figure 1: Firmware run-time state transition cycle

For instance, in the simplified UART ISR from Listing 3, the ISR *uart_tx_isr* makes the firmware continue running by changing the global object *output_buffer* status from full to not full. Besides, we observe that firmware widely uses specific instructions or infinite loops to enter a waiting state as well. Hence, we conclude that the firmware enters a waiting state if one of the following conditions is satisfied:

1. The firmware explicitly enables the global interrupt by setting the bit in CPSR in a frequent manner.
2. The firmware executes the *Wait for Interrupt* (WFI) or *Wait for Event* (WFE) instruction. These instructions allow the core to enter a low-power mode and stop executing code.
3. The firmware enters an infinite loop.
4. The firmware constantly checks the global objects whose value can be modified in an ISR.

With these assumptions in mind, we conclude that the firmware only requires interrupts when it is in a waiting state, and the ISRs will modify global objects to change the firmware run-time state from waiting to processing. Hence, we trigger interrupts in the following manner: When the firmware enters a waiting state via the first three conditions, we trigger all the effective interrupts one by one since we have no hints indicating which interrupt it requires. When the firmware enters a waiting state via the fourth condition, we trigger the corresponding interrupt whose ISR can change the global objects' value.

Our findings closely align with our investigation of firmware samples collected from various RTOS. More specifically, we analyzed 110 firmware samples and found that 83% of them followed our run-time state transition cycle observation. For more information about the investigation we conducted, please refer to Section 6.

5 Design and Implementation

We now discuss the threat model we use in this paper and then present the design and implementation of our approach.

5.1 Threat Model

In this paper, we assume that the attacker has full control over the MMIO data. The firmware accepts peripheral inputs (e.g., network packets, temperature, and console input characters) either from the MMIO registers or from the DMA buffer. We disregard the DMA input and focus on the data read from MMIO. We do not assume that the interrupts are controlled by the attacker, as they usually cannot be configured by an attacker. We make no assumptions about the image symbols, source code, and documentation of the firmware. However, like other re-hosting systems, we assume that we have full knowledge of the board's memory layout on which the firmware runs.

5.2 High-Level Overview

Figure 2 illustrates the design overview of AidFuzzer. It mainly consists of three components: the emulator, the IRQ modeling engine, and the fuzzing engine. The emulator maintains an emulated environment, including CPU registers and virtual memory, for re-hosting the firmware. It translates the ARM assembly code into native code and executes it. The IRQ manager triggers interrupts when the firmware enters the waiting state. The IRQ modeling engine extracts the firmware context and analyzes the ISR at a specific point. Upon completion of the analysis, the modeling results are saved in the IRQ model database, which is subsequently retrieved and used by the IRQ manager. The fuzzing engine, like other firmware fuzzers, supplies fuzzing input via MMIO and obtains coverage feedback from the emulator to guide the fuzzing process. In the following sections, we discuss the detailed functionalities of each part, in particular how the interrupt trigger mechanism works in the system.

5.3 IRQ Modeling Engine

Recall that we designate an IRQ as effective if it changes the firmware runtime state to processing (i.e., its ISR alters global objects), and we trigger it after its status becomes ready. The main goal of the IRQ modeling is to determine the type and

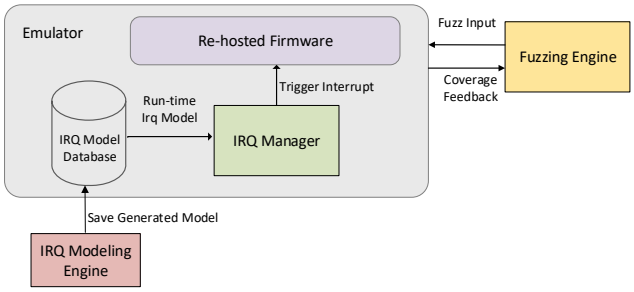

Figure 2: Design overview of AidFuzzer

status of an IRQ. The type and status can change during the runtime, therefore, we need to update the IRQ model when necessary. Specifically, we collect the following information during the IRQ modeling process.

a) Does the ISR modify any global objects? If so, collect the addresses to which the values are written.
b) Does the ISR use any function pointers? If so, collect the addresses from which the pointers are loaded.
c) Does the ISR de-reference any null pointers without checking? If so, collect the addresses from which the pointers are loaded.
d) Does the firmware always get stuck in the ISR, e.g., in an infinite loop?

We use the modified global objects information to identify the types of the IRQ (effective or ineffective), the null pointer de-reference and getting stuck information to identify the status of the IRQ (ready or unready), and the function pointer information to update the type and status of an IRQ. After finishing the modeling, the results are saved in the IRQ model database.

5.3.1 IRQ Modeling Workflow

The IRQ modeling engine takes firmware register values and a memory dump in combination with the memory layout configuration file as inputs (note that we assume that the memory layout is available in our threat model). The modeling engine symbolically executes the ISR by using angr [9]. During execution, the engine intercepts the memory access operations to collect the aforementioned information. However, since all data in the context dump is concrete, the global memory needs to be symbolized to explore as many paths as possible. An exception is the pointer, where further information is required to fetch more data and explore indirect functions. Specifically, we perform the following operations in the symbolic execution interception.

Interceptions. For memory read operations where the addresses from which the data is to be loaded are in the global memory space, the engine symbolizes their value and creates a mapping between the symbolic value and its concrete memory data. If the addresses are symbolic, it checks whether their corresponding concrete data is 0, and it reports a null pointer de-reference if the value is not constrained to a non-zero value. If a function pointer is used in an indirect branch, it resolves its symbolic value to concrete data so that the symbolic execution keeps running. If the function pointer is 0, it reports a function pointer usage. For memory write operation, it checks if the addresses are located in the global memory space, and if so, it reports a global object modification. For each piece of information, the engine collects the addresses at which the data was loaded or saved. After the symbolic execution is completed, it checks whether one of the paths can reach the end of the ISR; if not, it reports that it is stuck.

Special control register handling. The processor automatically loads the IRQ number into IPSR before entering the ISR. In certain firmware, a unified ISR wrapper is implemented for all IRQs, using the IPSR register value as an index to retrieve the actual ISR function pointer from a function pointer array. When conducting symbolic execution and encountering reads from this register, we provide the concrete IRQ number. For other control registers, we opt to symbolize their values to facilitate the exploration of additional execution paths.

Path explosion handling. To mitigate path explosion, we first explore the newly discovered basic blocks and set a timeout for the loops when modeling the ISR. We set a two minute timeout for the AidFuzzer prototype.

5.3.2 IRQ Modeling Example

We take the simplified UART ISR as a modeling example. During the symbolic execution, the entire global memory and MMIO data are symbolized. Therefore, the function *uart_ready* can return both true and false, and the ring buffer could also be empty and full. If one of the two conditions is unsatisfied, the execution ends, making it possible to reach the end of the ISR. If the two conditions are satisfied, the execution enters the `if` branch. When it tries to write the *output_buffer* object, the engine infers that it is a global object, and therefore it collects the address of the field that is being written to. Finally, we conclude that the UART ISR has the following modeling results: It modifies a global object *output_buffer*, it does not contain null pointer dereferencing, and does not cause the firmware to get stuck. Thus, the UART IRQ is effective and ready.

5.4 Emulator

We implement our emulator based on QEMU [10]. The emulator functions as a dynamic runtime environment for the firmware, with the original ARM assembly code dynamically compiled into intermediate language code (TCG code in QEMU). This intermediate code is further translated into native code and executed. The physical addresses of the re-hosted firmware are translated into native addresses through the use of the softMMU, enabling interception of every memory access. To feed the fuzz input, we implement an ARM Cortex-M-based board that supports full customization for the memory regions. Originally, the emulated peripherals are responsible for triggering interrupts; however, the IRQ manager takes over the interrupt triggering and can decide when and what interrupts are to be triggered with the help of the IRQ model database. After each fuzzing run, the emulator restores the re-hosted firmware and IRQ manager state. The coordination between the IRQ manager and the re-hosted firmware is explained in more detail next.

5.4.1 IRQ Manager

State monitor. Recall that we established four conditions, and if any of them is satisfied, the firmware enters the waiting state. Upon the satisfaction of any of these conditions, our callback function is invoked. In this callback function, the IRQ manager determines whether and what interrupts are to be triggered. We explain how AidFuzzer checks the satisfaction of the conditions and infers the firmware runtime state:

1. The firmware frequently enables the interrupt. We monitor the global interrupt enable/disable state by intercepting the execution of all *CPSIE I* instructions. This instruction sets the CPSR bit so that the processor can serve the interrupts. When writing to this register, our callback function is invoked.
2. The firmware executes WFI or WFE instructions. We intercept all the WFI and WFE instructions. Once the firmware executes these two instructions, the firmware stops execution and our callback function gets invoked.
3. The firmware enters an infinite loop. We search for all infinite loops in the firmware before fuzzing. Beginning with each branch instruction, we conduct symbolic execution of the subsequent instructions. If we determine that the execution can reach the same branch instruction without encountering an opportunity to exit the loop, we categorize it as an infinite loop. For each basic block initiating an infinite loop, we register a callback function. Consequently, should the firmware enter an infinite loop during runtime, the IRQ manager receives a notification.
4. The firmware constantly checks the global objects whose values can be modified in an ISR. We set memory read breakpoints to all the global objects whose values can be modified in the effective IRQs' ISRs. Whenever any of the global objects is read by the firmware, our callback functions are invoked.

We set a counter for each condition. If any callback function gets invoked, we increase the corresponding counter by one. Once a counter surpasses a predefined threshold, we trigger the specific interrupts. For example, we set the counter threshold for condition 1 to 10. If the firmware enables the interrupt 10 times, we trigger all the effective IRQs one by one and reset the counter to 0. Note that we trigger all the effective IRQs one by one when the first three condition counters surpass the threshold, while we only trigger the corresponding IRQ in the fourth condition. In AidFuzzer, we set the enable interrupt counter threshold to 32, the WFI/WFE instruction counter threshold to 1, the infinite loop counter threshold to 7, and the global object check counter threshold to 10 according to our empirical analysis.

To avoid recursive interrupt triggering, we do not increase the counter when the firmware is handling an exception, which means when the firmware is executing an ISR, even if the conditions are satisfied, no counter is increased.

IRQ triggering. Once the IRQ manager decides to trigger an interrupt, we set a bit in the NVIC IRQ pending field. Specifically, we call the QEMU function *armv7m_nvic_set_pending* with the IRQ number as an argument. The emulator checks the pending request and does the actual interrupt handling.

IRQ model switching. Maintaining the latest IRQ model is crucial. We achieve this by registering several event hooks to trigger model updating. Initially, we generate a model whose status is not ready for all the IRQs. During the fuzzing process, we analyze the ISR, save the results to the model database, and fetch the model when the currently used model needs to be updated. Specifically, we update the IRQ model in the following situations: a) When an IRQ is enabled. IRQ manager requests the IRQ modeling engine to analyze the newly enabled IRQ and switch the IRQ model to the generated one. b) When the vector table is re-based. We check all the enabled IRQ vector table entries to see if it is a new value. If so, the IRQ manager requests a re-analysis and switches to the new model. c) When an enabled IRQ table entry is overwritten with a new value. We re-analyze the ISR and switch to the new model. d) When a function pointer is overwritten with a new value. We re-analyze the ISR and switch to the new model. We assign a unique ID to each generated model according to its ISR address and the function pointer values. When an existing model is available in the database, we switch to the existing one instead of requesting a re-analysis.

5.4.2 Snapshot

We use snapshots to speed up the fuzzing process. When the firmware executes the first MMIO read instruction, we take a snapshot, as we rely on the assumption that the firmware's control flow will not change if it is not affected by the MMIO input. This holds for almost all the firmware, and it works well for all our test cases. Besides the memory and the registers, we snapshot and restore the internal NVIC state as well.

5.5 Fuzzing Engine

Multi-stream fuzzing input. We adopt Hoedur's [4] multi-stream input and Fuzzware's fine-grained input model [3]. Whenever the firmware tries to read from MMIO memory, the emulator generates a corresponding ID by calculating the instruction and MMIO address hash result. If it is the first time it encounters the access ID, it invokes the Fuzzware interface to generate an input model for the ID, then the emulator notifies the fuzzer about the newly generated input stream. For each newly generated stream, the fuzzing engine assigns a random length of data for it. The firmware consumes the stream data from the MMIO read when the stream is not exhausted; otherwise, it reports an out-of-stream exit. For now, we ignore the MMIO writes and redirect them to a dummy function.

Coverage feedback. Edge coverage is widely adopted in fuzzing [11] [12] [13]; however, the asynchronous interrupt leads to noisy coverage. An edge that starts from the current basic block to the beginning of the ISR does not exist. To eliminate the noise, we choose to use basic block coverage. We intercept every basic block execution. Before the basic block gets executed, we increase the corresponding coverage byte by one.

Crash detection. We do not have sanitizers integrated into our emulator; therefore, we only detect invalid memory access crashes. When an invalid memory access happens, such as a null pointer de-reference when address 0 is not mapped, our exception hook is notified and gets called with the exception code as an argument. We check the exception code to see if it is a real crash since, for example, a syscall is also regarded as an exception in ARM Cortex-M. Moreover, the firmware may write to an NVIC field to reset the system. Fuzzware regards this as a crash; however, we filter out such cases since they do not incur a security issue.

6 Evaluation

In this section, we comprehensively evaluate AidFuzzer, demonstrating its effectiveness on firmware fuzzing. We aim to answer the following research questions.

RQ1: Is AidFuzzer more effective compared to the previous works for fuzzing firmware in terms of coverage and bug finding?

RQ2: How sound is the IRQ modeling?

RQ3: How computationally expensive is the implemented IRQ modeling?

RQ4: Does the IRQ modeling perform better than existing methods?

6.1 Experiment Setup

Experiment settings. We performed our experiments on a 104 core Intel Xeon Gold 5320 CPU @ 2.20GHz with a 252 GB RAM server running a Ubuntu 22.04.1 LTS OS. We evaluated our prototype against the two state-of-the-art firmware fuzzers Fuzzware and Hoedur. For each target, we gave each fuzzer one physical CPU core. Moreover, we evaluated against SafireFuzz.

Target firmware selection. Our evaluation targets consist of 10 firmware projects. We collected them from open-source GitHub projects, well-known RTOS examples, and the targets that have been used by previous evaluation experiments [2]. The details can be found in Table 1.

Evaluation metrics. We evaluate the effectiveness of AidFuzzer in two aspects: a) We count the number of unique basic blocks discovered by fuzzers. We measure if AidFuzzer can discover more unique basic blocks or can discover basic blocks faster. b) We count the number of discovered unique crashes and the number of confirmed vulnerabilities. We measure if AidFuzzer can discover more vulnerabilities while having fewer false positives.

[1]Microchip Advanced Software Framework

Table 1: Number of basic blocks, board, and the OS/framework information of the firmware we collected.

	Bbls	Board	OS/framework
Blehci [14]	5441	nrf52840	Apache Mynewt
AnnePro2-Shine [15]	1117	AnnePro2 keyboard	ChibiOS
TauLabs [16]	4644	pipxtreme	ChibiOS
3Dprinter [17]	8032	Marlin printers	bare-metal
bcn_rfd_ncp [18]	3590	Atmel SAM	ASF[1]
coord_ncp [18]	4247	Atmel SAM	ASF
mac_no_beacon_sleep [18]	2940	Atmel SAM	ASF
nobcn_ffd_ncp [18]	3510	Atmel SAM	ASF
sam4l_qtouch [18]	1799	Atmel SAM	ASF
nmea_parser [19]	8415	STM32F411RE	RT-Thread

6.2 Effectiveness of AidFuzzer (RQ1)

We compare AidFuzzer against the two state-of-the-art tools Fuzzware and Hoedur with their advanced fuzz-mode interrupt triggering mechanism integrated. Each is configured with the default 1000 basic blocks interval. To eliminate the effects originating from the fuzzer, we implemented a fuzz-mode interrupt triggering for AidFuzzer as well for comparison. The implemented AidFuzzer-fuzz-mode has the same interrupt triggering settings, such as interval, as Fuzzware and Hoedur. We fuzzed each target for 24 hours 10 times as recommended by Klees et al. [20] and Schloegel et al. [21]. Figure 3 visualizes the median and confidence interval of discovered basic blocks, and Table 2 presents the number of unique reported crashes and the confirmed vulnerabilities.

6.2.1 Coverage Analysis

As shown in Figure 3, AidFuzzer achieved higher and faster coverage than Fuzzware, Hoedur, and AidFuzzer-fuzz-mode for the majority of the targets, while for other targets, AnnePro2-Shine, nmea_parser, and sam4l_qtouch, AidFuzzer achieved similar coverage.

Hoedur had a bug when handling a UART interrupt priority in Blehci and could not continue before the firmware started processing data. For TauLabs, due to ineffective IRQs, Hoedur triggered the watchdog interrupt and got stuck in an infinite loop, and no fuzzing progress was made. We did not plot Fuzzware and AidFuzzer-fuzz-mode for TauLabs, as both fuzzers crashed before discovering any valid input due to triggering unready interrupts. The same problem also happened for target nmea_parser. The nmea_parser SysTick interrupt ISR used uninitialized pointers, which was triggered by Fuzzware and AidFuzzer-fuzz-mode in the early stage. For sam4l_qtouch, AidFuzzer-fuzz-mode kept triggering the SysTick interrupt whose ISR involves an infinite loop before the second interrupt was enabled. An interesting target

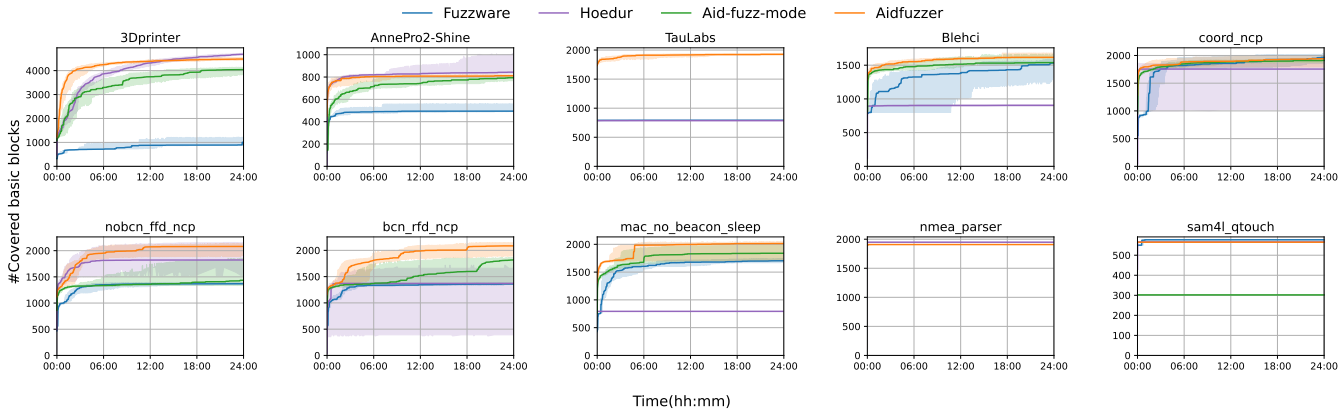

Figure 3: Basic block coverage achieved by Fuzzware, Hoedur, AidFuzzer-fuzz-mode, and AidFuzzer over the course of 24 hours for 10 times. We plot the median and confidence interval.

is mac_no_beacon_sleep. As recommended by Hoedur, we disabled the interrupt triggering time interval because the firmware contains WFI instructions(which means Hoedur only triggers an interrupt when it encounters a WFI instruction). However, the firmware did not reach the instruction during the execution. We identified four conditions that can make the firmware enter a waiting state. However, Hoedur cannot fully identify all of these conditions.

As shown in Figure 3, although the interrupts did not make the firmware get stuck or crash in the early stage, triggering the interrupts in a proper frequency was crucial for firmware fuzzing as well and made the fuzzer achieve faster basic block coverage. We take the 3Dprinter as an example to illustrate the reason behind it.

The 3Dprinter firmware takes a string—called GCode instruction—as input. The GCode instruction is read one character at a time when the UART interrupt is triggered. This character is stored in a buffer and is only processed later on when a whole line is read. For the fuzz-mode interrupt triggering strategy, it can be an issue to associate the coverage of the GCode instruction execution with triggering the UART interrupt multiple times in combination with a meaningful GCode instruction input. A fuzzer with such an interrupt strategy may only rarely raise the UART interrupt due to the mentioned coverage feedback disconnect and therefore greatly decreases the chance of reaching deep into the 3Dprinter logic. AidFuzzer identifies the UART interrupt as an effective and ready IRQ. Its ISR modifies the number of characters stored in the buffer. The firmware keeps checking the number of elements in the buffer when it does not receive enough characters from the UART. AidFuzzer's state monitor recognizes the firmware run-time state as waiting and triggers the UART interrupt accordingly. This way, it increases the chances for the firmware to receive a whole line of GCode and continue processing. We noticed that Hoedur achieved higher coverage than AidFuzzer after 14 hours. We found that an interrupt was not triggered by AidFuzzer. This interrupt is enabled after a failure occurs and the firmware enters a throb function. The interrupt is not used by the firmware. AidFuzzer identifies the IRQ as unready as it contains uninitialized pointers that persist throughout the throb function, thus AidFuzzer did not trigger this interrupt, incurring lower coverage than Hoedur.

Besides Fuzzware and Hoedur, we also compared AidFuzzer to SafireFuzz using the 12 samples from the SafireFuzz experiments [5]. Since SafireFuzz requires manual HAL function hook, we reused the data from their paper and plot them separately. Figure 4 shows the unique basic blocks discovered by AidFuzzer, AidFuzzer-fuzz-mode, and SafireFuzz. Notably, six of the firmware samples (6LoWPAN_Receiver, 6LoWPAN_Transmitter, P2IM_Drone, P2IM_PLC, STM_PLC, WYCINWYC) do not use DMA to transfer data, hence these samples are fully supported by AidFuzzer. The other samples use DMA, which is an orthogonal challenge not addressed by AidFuzzer. In the WYCINWYC, P2IM_Drone, 6LoWPAN_Transmitter, and P2IM_PLC samples, AidFuzzer achieves similar or better coverage compared to SafireFuzz. However, AidFuzzer discovered fewer unique basic blocks than SafireFuzz in STM_PLC and 6LoWPAN_Receiver. We observed that the STM_PLC requires a nested interrupt to be triggered, which AidFuzzer does not support. Additionally, AidFuzzer could not successfully recognize the global objects in 6LoWPAN_Receiver due to a bottleneck in symbolic execution. We emphasize that while AidFuzzer and SafireFuzz address orthogonal firmware fuzzing challenges, our methodology could be adapted to enhance the fuzzing efficiency of SafireFuzz.

6.2.2 Crash Analysis

AidFuzzer found in a total of 8 vulnerabilities in the 10 firmware targets shown in Table 2, including 1 buffer over-read control flow hijacking in TauLabs, 3 buffer

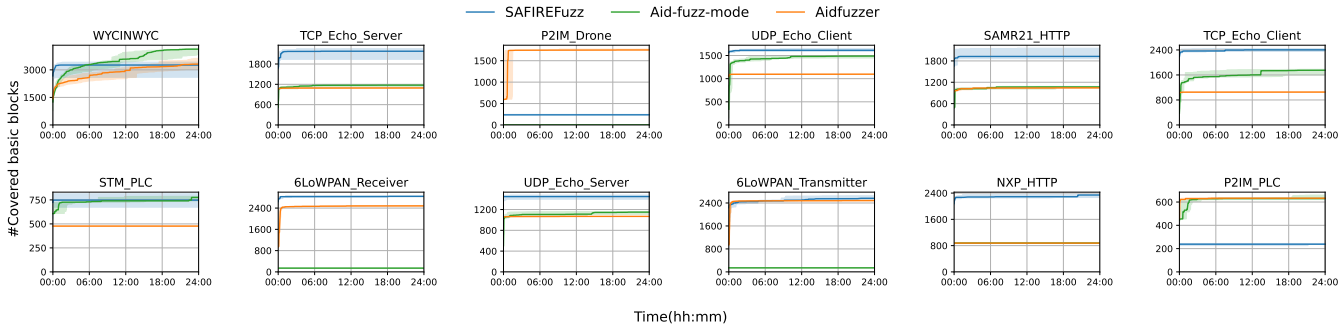

Figure 4: Basic block coverage achieved by SafireFuzz, AidFuzzer-fuzz-mode, and AidFuzzer over the course of 24 hours for 5 times. We plot the median and confidence interval. AidFuzzer and SafireFuzz work on different architectures; therefore, we reused the evaluation data from the SafireFuzz experiments.

Table 2: Unique crashes, confirmed vulnerabilities, and vulnerability types found by Fuzzware, Hoedur, AidFuzzer-fuzz-mode and AidFuzzer. AidFuzzer discovered more vulnerabilities while reporting 0 false positives.

	Fuzzware		Hoedur		Aid-fuzz-mode		AidFuzzer		
	Reported	Confirmed	Reported	Confirmed	Reported	Confirmed	Reported	Confirmed	Vulnerability types
Blehci	1	0	0	0	0	0	0	0	
AnnePro2-Shine	3	3	3	3	3	3	3	3	buffer over-write
TauLabs	1	0	0	0	0	0	1	1	buffer over-read
3Dprinter	1	0	1	0	1	0	0	0	
bcn_rfd_ncp	1	1	1	1	1	1	1	1	arbitrary write
coord_ncp	1	1	1	1	1	1	1	1	arbitrary write
mac_no_beacon_sleep	1	1	0	0	1	1	1	1	arbitrary write
nobcn_ffd_ncp	1	1	1	1	1	1	1	1	arbitrary write
sam4l_qtouch	0	0	0	0	0	0	0	0	
nmea_parser	1	0	1	0	1	0	0	0	
total	11	7	8	6	9	7	**8**	**8**	

over-writes control flow hijacking in AnnePro2-Shine, and 4 arbitrary memory writes in bcn_rfd_ncp, coord_ncp, mac_no_beacon_sleep, and nobcn_ffd_ncp. It is worth noting that AidFuzzer did not report any false positives. Hoedur, Fuzzware, and AidFuzzer-fuzz-mode correctly reported part of the vulnerabilities; however, they reported false positives as well. We reported all the vulnerabilities to the vendors.

Fuzzware misreported a reset in Blehci as a crash. Due to the implementation bug, Hoedur did not correctly handle the interrupt priority in Blehci and therefore got stuck. Fuzzware, Hoedur, and AidFuzzer-fuzz-mode reported null pointer dereferences in 3Dprinter. We manually checked the firmware, and we found that the crashes all happened in an ISR which has not been fully initialized. When a failure occurs, this IRQ is enabled by accident and should not be triggered in the real environment. AidFuzzer successfully identified it as an unready IRQ and did not trigger it. Fuzzware, Hoedur, and AidFuzzer-fuzz-mode all got stuck in the early stage when fuzzing the TauLabs firmware due to ineffective and unready IRQs. AidFuzzer avoided triggering the IRQs that cause the firmware to get stuck and successfully found the buffer over-read vulnerabilities which were not covered by other fuzzers.

AidFuzzer successfully found all the vulnerabilities that the state-of-the-art tools could also find. Moreover, AidFuzzer found more vulnerabilities that were not found by others and reported fewer false positives, which saves manual effort for crash analysis. Note that we only counted the number of unique crashes. Hoedur and Fuzzware reported a large number of false positive crashes in these targets which is why verification requires a non-trivial manual effort. Five CVEs have been assigned to our findings.

6.3 Soundness of IRQ Modeling (RQ2)

We collected the following IRQ information for each firmware presented in Table 3: the number of enabled IRQs, the number of unique ISRs, the number of effective ISRs (effective ISR means the ISR makes the corresponding IRQ effective), the number of monitored global objects, the number of null data pointers, the number of function pointers, and the mechanisms employed by the firmware to enter the waiting state. The presented table reveals that firmware deploy complex interrupt services for their functionalities. A portion of the ISRs renders the corresponding IRQs ineffective. The range of monitored global objects spans from 5 to 186. The number of function pointers and null-data pointers is consistently below 50 for all targets. Regarding the conditions to enter the waiting state, the firmware utilizes all mentioned four conditions. However, the preference for specific methods varies depending on the implementation of each firmware.

Table 3: IRQ Modeling Result. Effective ISRs refer to the ISRs that make the corresponding IRQ effective. Global objects refer to the global objects that the ISRs modify. We count the overall numbers for all enabled IRQs in the firmware. The methods to enter the waiting state are: ❶ constantly enable global interrupt, ❷ execute WFI/WFE instructions, ❸ infinite loop, ❹ constantly check global objects.

	# of enabled IRQs	# of unique ISRs	# of effective ISRs	# of global objects	# of NULL data pointers	# of function pointers	enter waiting state
Blehci	7	10	6	84	4	4	❷❶
AnnePro2-Shine	3	3	3	57	5	1	❸
TauLabs	10	10	8	186	13	11	❸❹
3Dprinter	7	7	6	66	3	13	❹
bcn_rfd_ncp	4	5	3	11	0	16	❶❹
coord_ncp	4	4	3	11	0	15	❶❹
mac_no_beacon_sleep	3	3	2	16	0	45	❶❹
nobcn_ffd_ncp	4	4	3	13	0	19	❶❹
sam4l_qtouch	3	3	2	5	0	1	❶❹
nmea_parser	2	2	2	45	4	25	❶❹

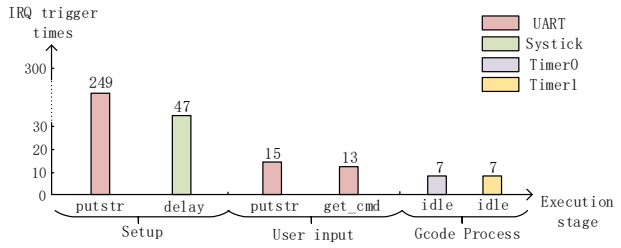

Figure 5: Interrupt triggering in 3Dprinter

6.3.1 Ineffective IRQ Case Study

Taking the Blehci firmware as an example, it enables 7 IRQs and employs 10 ISRs to handle these IRQs. However, 4 of these ISRs render their corresponding IRQs ineffective. One such IRQ is associated with the watchdog. The implementation of the watchdog ISR involves halting the system by executing a break-point instruction and then entering an infinite loop. If triggered, this interrupt results in the termination of the fuzzing process. Another interrupt is tied to the SysTick, and its ISR is an infinite loop. Activation of this interrupt also leads to the cessation of the fuzzing process. Our IRQ modeling result correctly identifies the watchdog and SysTick IRQ as ineffective.

6.3.2 Effective IRQ Case Study

Manually verifying the soundness of each modeling result for effective IRQs in the target set can be a cumbersome task. Nonetheless, it is worth illustrating the result with the example of the 3Dprinter. We manually analyzed the firmware code logic and understood the ISR functionalities. Then we checked if the modeling results fit our manual analysis results. To have a clear representation of the AidFuzzer interrupt-triggering for 3Dprinter, we executed the firmware with a discovered input and visualized the type and frequency of interrupts triggered during the execution, as depicted in Figure 5. The x-axis is the firmware execution stage, the differently colored columns represent the number and type of the interrupts triggered. We mark the function names where the interrupts get triggered for intuitive understanding.

In the 3Dprinter firmware program logic, IRQ 15 serves the SysTick, and its ISR increases a counter by one. IRQ 53 serves an UART device, and its ISR either consumes one character from the output buffer or reads a character from the UART register into the input buffer. IRQ 44 and 66 serve two timers.

In the setup stage, the UART interrupt is only triggered in the function *usart_putstr*. This function keeps looping until all the characters are consumed. AidFuzzer triggers the UART interrupt to consume the output buffer and lets the firmware continue running. The SysTick interrupt is only triggered in function *delay*. This function is used to set up the temperature management environment and it checks whether the counter exceeds a limit and then continues execution. AidFuzzer triggers the SysTick interrupt to increase the counter and therefore bypass the check quickly. In the user input stage, besides being triggered in the *usart_putstr* function, the UART interrupt is also triggered in function *GCodeQueue_get_serial_commands*. This function checks if there are enough characters in the input buffer and retrieves the characters to a command buffer. During this stage, the firmware reads the input from UART, and thus AidFuzzer triggers the UART interrupt to fill the input buffer. In the last Gcode process stage, two timer interrupts are triggered in the function *idle*. In this stage, the firmware is busy processing the user input, therefore AidFuzzer does not trigger any other interrupts. The firmware waits for some tasks to be completed; consequently, AidFuzzer triggers the timer interrupt to notify the firmware about the completion of the task. The overall IRQ modeling result fits our manual analysis result well.

6.3.3 Heuristic Study

In order to verify if our firmware run-time transition cycle applies to the majority of the firmware, we conducted

a heuristic study. Analyzing firmware binaries without access to the source code requires considerable manual effort and is prone to errors. Given that firmware is typically closed-source, we collected 110 firmware samples from well-known open-source RTOS examples. As shown in Table 6, we manually analyzed their implementation logic and the corresponding models based on our observations. We found that 19 (17%) of the samples do not perform the run-time transition cycle discussed in our paper, while 91 (83%) adhere closely to our heuristics: The firmware performs a waiting-processing run-time state transition cycle. For the 91 samples that follow our heuristics, we analyzed the functions used for the waiting and processing logic, the interrupt service routines, and the global objects involved in changing the run-time state. Regarding the methods used to transition to the waiting state, 7 (6%), 19 (17%), 8 (7%), and 82 (74%) of the samples use one of the four identified methods, respectively. Our results indicate that continuous checking of global objects is a common method in the analyzed firmware samples. However, the use of different methods to enter the waiting state is mostly independent of each other and highly dependent on the RTOS design logic. For instance, the ChibiOS samples exclusively use an infinite loop, a method that is not commonly used. From this heuristic study, we conclude that our observation applies to the majority of firmware samples.

We proposed four conditions that a firmware can use to enter the waiting state and applied empirical counter values to each threshold for the four conditions. Depending on the design logic and implementation of the firmware, the values we have chosen may not be optimal. For example, in the case of the 3D printer firmware, which uses two fixed-length buffers to send and receive characters over UART, both reading and sending characters take place within a single ISR. The ISR checks the value of the UART register to determine readiness for sending or receiving characters. When the sending buffer is full, the firmware continuously checks the buffer status until it is no longer full, then writes the character to the buffer. Compared to the sending request, the reading request is less frequent. If the threshold for checking global objects is set too low, the ISR will be triggered excessively, making it easier to consume characters from the sending buffer, but also increasing the data read for processing, thereby expanding the input space. Conversely, if the threshold is set too high, the ISR will be triggered infrequently, reducing the input space but failing to meet the character-sending requests. This can lead to the coverage feedback being interrupted if loops are executed without new basic blocks being detected.

6.4 Overhead (RQ3)

The additional overhead primarily stems from three sources. The first source is the memory read/write breakpoints. Despite our efforts to optimize the code and minimize the impact, it still incurs a 20%-25% overhead throughout the entire fuzzing process, as every memory access in the firmware undergoes scrutiny.

Table 4: Extra overhead caused by IRQ modeling

	Infinite loop searching (s)	IRQ Modeling times	Total time consuming (s)
Blehci	133	11	525
AnnePro2-Shine	32	7	286
TauLabs	51	15	1433
3Dprinter	136	7	397
bcn_rfd_ncp	76	7	524
coord_ncp	83	7	508
mac_no_beacon_sleep	66	6	390
nobcn_rfd	64	7	519
sam4l_qtouch	51	3	6
nmea_parser	232	4	489

Table 5: IRQ modeling comparison with AIM. *Ident* refers to the global objects identified by the tool. *TP* refers to the correctly identified global objects.

	AIM		AidFuzzer		Time(s)	
	Ident	TP	Ident	TP	AIM	AidFuzzer
cnc_r1	1	1	20	15	28	126
gateway_r2	13	13	12	12	1270	375
plc_r1	13	13	11	11	590	71
robot_r1_hardfpu	1	1	19	19	1784	117
reflow_oven_r1	13	13	1	1	699	135

The second overhead arises from the search for infinite loops in the firmware image. Table 4 details the time consumed by AidFuzzer in locating infinite loops for all targets. The majority of these searches can be completed within 250 seconds. It is important to note that we conducted this search only once for each target, and the results can be reused in subsequent fuzzing.

The third contributor to overhead is the IRQ modeling. When the IRQ model needs to be updated and the model is not found in the model database, an IRQ modeling is conducted. Table 4 provides the overall times and time used for the IRQ modeling. In our experiments, IRQ modeling occurred mostly in the initial half-hour, and the modeling results were reused in subsequent fuzzing. Therefore, the overhead associated with IRQ modeling is deemed acceptable when compared to the overall fuzzing time.

6.5 IRQ Modeling Comparison with AIM (RQ4)

Since AidFuzzer and AIM use different underlying methods to analyze firmware—fuzzing and symbolic execution, respectively—a direct comparison of the number of discovered basic blocks is not meaningful. Instead, we conducted a quantitative comparison of AidFuzzer and AIM in terms of IRQ modeling in the firmware samples analyzed in the AIM paper experiment [6]. Both AidFuzzer and AIM share the insight that firmware changes global objects in the ISR to change execution behavior. Therefore, we counted the number of unique global objects identified by both methods and manually inspected their correctness as well as the analysis

time spent on IRQ modeling. As shown in Table 5, AidFuzzer and AIM both identified the global objects correctly. However, AidFuzzer reported five false positives in cnc_r1. AidFuzzer aims to find the global objects that can be modified as many as possible. Therefore, AidFuzzer tries to symbolize the variable values thus it can explore all possible paths. As a result, due to the symbolic execution mechanism, some unreachable paths can be explored by our modeling engine, and the corresponding global objects that are modified within the paths are incorrectly identified. This comprehensive path exploration can lead to false positive global objects that cannot change the execution behavior of the firmware. The time required for IRQ modeling varied between a few seconds and minutes for the two methods, depending on the firmware logic. These variations in AidFuzzer are acceptable, as the modeling process only occurs once during fuzzing and can be reused in subsequent fuzzing runs. We observed that AidFuzzer in general spent less time on the IRQ modeling compared to AIM. More specifically, AidFuzzer only spent an average of 33% of the analysis time on the five samples used in the AIM experiments. For cnc_r1, AidFuzzer spent more time on analysis than AIM. The reason is that a register indicates the status of the device for a TIM IRQ ISR. While AidFuzzer explores all possible paths by symbolizing the register values, AIM only analyzes one path by giving the register a concrete value.

7 Related Work

Coverage-guided fuzzing works [11] [13] [12] [22] have found a tremendous number of vulnerabilities in the past decades. To achieve higher coverage, lightweight Redqueen [23], IJON [24], and heavy-weight symbolic execution are used.

Nowadays, besides fuzzing the general programs, domain-specific fuzzing is becoming more and more popular and has gained much attention, such as kernel fuzzing [25] [26], system-wide hypervisor-based fuzzing [27] [28], drone fuzzing [29], hypervisor fuzzing [30] [31] [32] [33] [34] and trusted execution environment fuzzing [35] [36]. To fuzz specific programs, the target-specific problems need to be solved first. For example, when fuzzing the trusted execution environment, the memory cannot be directly accessed during the run-time Therefore, previous works found methods to get the coverage or other feedback to guide fuzzing.

In this paper, we focus on embedded device firmware fuzzing. Instead of re-hosting the firmware in an emulator environment, black-box fuzzing tools, such as Iotfuzzer [37], feed the fuzzing data from real devices (e.g., mobile phone Apps). Although black-box fuzzing mitigates the cumbersome effort to set up the emulator environment, without having direct access to the firmware memory, it suffers from no coverage guidance. The same problem also happens to drone fuzzing [29] and [36].

Semi-simulation approaches [38] [38] [39] [40] [41] [42] [43] implement a hardware in the loop method to forward the hardware access to the real physical devices, while the firmware itself runs in a simulated environment. This approach, however, requires lots of physical devices. Besides, as the generation of the data from physical devices is slow and cannot be easily controlled, this approach is not suitable for fuzzing. Due to the presence of physical devices, it is difficult to deploy a parallel analysis.

Running the whole firmware in an emulator environment enables direct memory access to the firmware, turning it into a grey-box fuzzing. Qemu [10] has been widely used in full-system emulation. Qemu-based fuzzers [44] [45] [46] [47] heavily rely on the target-specific information that a large corpus of general firmware cannot deploy. HALucinator [48] and [49] propose to identify the hardware abstraction layer in the firmware for re-hosting. Unfortunately, the hardware abstraction layer has not been widely adopted by firmware development yet. Modeling the hardware abstract layer still requires much manual effort such as reverse engineering. Without making too many knowledge assumptions about the firmware, full system emulation-based fuzzing is more scalable and requires less manual effort. PRETENDER [50] runs the original firmware independently in a simulated CPU. However, it still needs the hardware to model the peripheral behaviors. The peripheral behaviors are then solved by P2IM [1] by extracting the knowledge from the documentation or manuals and later further addressed by μEmu [2] and Fuzzerware [3] by symbolic execution modeling. Extracting the peripheral behavior model from the documentation is not stable and sometimes not reliable. Symbolic execution is known to be limited by its low scalability and confined to a small control flow scope. For example, it cannot model the MMIO data that is copied to global variables. Hoedur [4] splits the single-stream fuzzing data into multi-stream data that is identified by its MMIO address and its instruction address, making the fuzzing data field aware. It avoids the avalanche caused by asynchronous interrupts. Recently, SafireFuzz [5] proposed to use dynamic binary re-writing to run the firmware on high-performance hardware. However, the hardware abstraction layer (HAL) needs to be present. Although HAL is becoming more and more popular in firmware development, there is still a large number of firmware that does not support it. In fact, only two of the samples in our dataset support HAL. In addition, manually hooking HAL requires a lot of manual effort, which is also error-prone. All the full system emulation-based fuzzing either implement a simple round-robin or fuzz-mode interrupt triggering mechanism or rely on the unstable model extracted from the documentation and, therefore, cannot handle the complex interrupt situation.

Concurrent to our work, AIM [6] proposed an idea similar to AidFuzzer for modeling firmware interrupts. However, our method has several advantages compared to AIM: First, AidFuzzer systematically investigates the relationship between

interrupts and firmware run-time state. The global variables bridge them. In contrast, AIM does not recognize the firmware run-time state in a general and high-level view. Second, we propose four conditions that the firmware can use to require an interrupt. In contrast, AIM considers only one of them. Third, AIM does not consider the interrupt status and, thus, cannot prevent the firmware from crashing prematurely. Finally, AIM has to analyze the ISR by using dynamic symbolic execution when an event is enabled, which means that it has to perform symbolic execution frequently. Moreover, its firmware emulation is based on symbolic execution. The overall design has a strong impact on the execution speed.

8 Discussion

Although the majority of the firmware follows the run-time state proposed in our paper, according to the heuristic study result, there is still a portion of firmware that does not align with this assumption. Even though some firmware does not rely on the interrupt to accomplish its tasks, accurately modeling the firmware interrupt in a broad range requires additional work that depends on a more generic assumption.

AidFuzzer triggers interrupts only after their data are fully initialized in the ISR. Some may consider crashes caused by uninitialized data as bugs. While AidFuzzer may lead to false negatives in such scenarios, we assert that these bugs are less closely tied to security vulnerabilities. Our primary focus is on fuzzing the deep logic within the firmware, prompting us to strike a balance between deep logic exploration and the potential for false negatives. Although certain ineffective interrupts are not triggered by AidFuzzer, resulting in slightly lower coverage, this impact is deemed trivial when compared to overall coverage.

Given the well-known bottleneck associated with symbolic execution, AidFuzzer faces challenges in effectively handling loops and intricate mathematical operations. To address issues related to control flow explosion, we chose to first explore the newly discovered basic blocks, albeit at the expense of potential false negatives. Moreover, when handling complicated nested structures, AidFuzzer cannot fully model the ISR, which may generate an incorrect result. When a data pointer is updated in the ISR that may alter the model result, we do not re-analyze it, resulting in an incorrect result as well. However, we need to make a trade-off between the fuzzing performance and the modeling soundness.

Compared with round-robin and fuzz-mode interrupt triggering mechanisms, AidFuzzer's adaptive method outperforms them in fuzzing, but may not correspond to real situations. For instance, the SysTick interrupt is only triggered at a fixed time interval in a real device, however, it is triggered multiple times within a short time window in fuzzing. We urge that as long as the discovered vulnerabilities can be reproduced in real devices, we can prioritize the fuzzing effectiveness.

9 Conclusion

In this paper, we propose AidFuzzer, an adaptive interrupt-driven firmware fuzzing prototype. By identifying the status and the type of the IRQ, we trigger the interrupts when the firmware enters the waiting state and leave the firmware running in the processing state. In our collected firmware targets, AidFuzzer achieved higher and faster coverage when dealing with complex interrupt firmware compared with the state-of-the-art approaches. Besides, we found eight previously unknown security issues in real-world firmware.

10 Acknowledgement

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research (BMBF) under the grant CPSec (16KIS1899).

References

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

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

[3] 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*, 2022.

[4] 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*, 2023.

[5] Lukas Seidel, Dominik Maier, and Marius Muench. Forming faster firmware fuzzers. In *USENIX Security Symposium*, 2023.

[6] Bo Feng, Meng Luo, Changming Liu, Long Lu, and Engin Kirda. AIM: Automatic Interrupt Modeling for Dynamic Firmware Analysis. In *Conference on Dependable Systems and Networks (DSN)*, 2023.

[7] Nested Vectored Interrupt Controller. https://developer.arm.com/documentation/ddi0439/b/Nested-Vectored-Interrupt-Controller.

[8] 8250 UART Programming Serial Programming. https://en.wikibooks.org/wiki/SerialProgramming/8250_UARTProgramming.

[9] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, Jessie Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: State of The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)*, 2005.

[11] american fuzzy lop. https://github.com/google/AFL.

[12] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[13] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[14] Apache blehci example. https://mynewt.apache.org/v1_5_0/tutorials/ble/blehci_project.html.

[15] Anne pro 2 shine! https://github.com/OpenAnnePro/AnnePro2-Shine/tree/master.

[16] high quality open source code for autopilots. https://github.com/TauLabs/TauLabs.

[17] Marlin firmware. https://marlinfw.org/.

[18] Advanced software framework. https://asf.microchip.com/.

[19] A tiny and elegant iot operating system. https://www.rt-thread.io//.

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

[21] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

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

[23] 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.

[24] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[25] syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[26] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[27] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Worner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *USENIX Security Symposium*, 2021.

[28] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[29] Nico Schiller, Merlin Chlosta, Moritz Schloegel, Nils Bars, Thorsten Eisenhofer, Tobias Scharnowski, Felix Domke, Lea Schonherr, and Thorsten Holz. Drone Security and the Mysterious Case of DJI's DroneID. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[30] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. HYPERFUZZER: An efficient hybrid fuzzer for virtual cpus. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[31] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *USENIX Security Symposium*, 2022.

[32] GaoningPan XingweiLin XuhongZhang Yongkang Jia and ShoulingJi ChunmingWu XinleiYing JiashuiWang YanjunWu. V-SHUTTLE: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[33] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Worner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[34] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. MORPHUZZ: Bending input space to fuzz virtual devices. In *USENIX Security Symposium*, 2022.

[35] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing. In *USENIX Security Symposium*, 2022.

[36] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, et al. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[37] 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.

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

[39] 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.

[40] 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.

[41] Marius Muench, Dario Nisi, Aurelien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[42] 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.

[43] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[44] 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.

[45] Andrei Costin, Apostolis Zarras, and Aurelien 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.

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

[47] 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.

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

[49] 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.

[50] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

Appendix

Table 6: Firmware model analysis based on the observation. *Method* refers to the four methods above to enter the waiting state. *F1* refers to the waiting function. *F2* refers to the processing function. *F3* refers to the interrupt service routine that causes the state transition. *GO* refers to the global object to be changed. Empty cells refer to the firmware that does not follow our model.

RTOS	Category	Firmware	Method	F1	F2	F3	GO
Zephyr	Network	echo	❶	while loop in main	z_impl_zsock_sendto	nrfx_gpiote_0_irq_handler	m_cb
		echo_async	❹	while loop in main	setblocking, z_impl_zsock_sendto	nrfx_gpiote_0_irq_handler	m_cb
		echo_async_select	❹	while loop in main	read, setblocking, write	nrfx_gpiote_0_irq_handler	m_cb
		dumb_http_server	❹	while loop in main	z_impl_zsock_recvfrom, z_impl_zsock_sendto	nrfx_gpiote_0_irq_handler	m_cb
		socketpair	❹	while loop in main	fun	nrfx_gpiote_0_irq_handler	m_cb
		mqtt_publisher	❹	while loop in main	mqtt_ping, publish	nrfx_gpiote_0_irq_handler	m_cb
		mdns_responder	❹	while loop in service	z_impl_zsock_socket, z_impl_zsock_bind	nrfx_gpiote_0_irq_handler	m_cb
	Bluetooth	central	❶❹	start_scan	bt_conn_le_create	radio_nrf5_isr	isr_cb_param
		broadcast_audio_sink	❷	loop in main	bt_bap_broadcast_sink_sync	radio_nrf5_isr	isr_cb_param
		bthome_sensor_template	❹	loop in bt_le_adv_update_data	get_adv_name_type, le_adv_update	radio_nrf5_isr	isr_cb_param
		broadcast_audio_sink	❹	loop in main	bt_bap_broadcast_sink_sync	radio_nrf5_isr	isr_cb_param
		central_gatt_write	❹	loop in central_gatt_write	write_cmd, bt_conn_unref	radio_nrf5_isr	isr_cb_param
		central_iso	❹	start_scan	handler	radio_nrf5_isr	isr_cb_param
		iso_broadcast	❹	loop in main	net_buf_simple_add_mem, bt_iso_chan_send	radio_nrf5_isr	isr_cb_param
		iso_receive	❹	loop in main	gpio_pin_set_dt	radio_nrf5_isr	isr_cb_param
		periodic_adv	❹	loop in main	bt_le_per_adv_set_data	radio_nrf5_isr	isr_cb_param
		periodic_adv_conn	❹	loop in main	z_impl_k_sleep	radio_nrf5_isr	isr_cb_param
		periodic_async	❹	loop in main	gpio_pin_set	radio_nrf5_isr	isr_cb_param
		periodic_sync_conn	❹	loop in main	bt_conn_disconnect	radio_nrf5_isr	isr_cb_param
		peripheral_gatt_write	❹	loop in peripheral_gatt_write	write_cmd, bt_conn_unref	radio_nrf5_isr	isr_cb_param
		scan_adv	❹	loop in main	z_impl_k_sleep, bt_le_adv_stop	radio_nrf5_isr	isr_cb_param
		tmap_bmr	❹	loop in main	bt_bap_broadcast_sink_sync	radio_nrf5_isr	isr_cb_param
		unicast_audio_client	❹	loop in main	configure_stream, printk	radio_nrf5_isr	isr_cb_param
		unicast_audio_server	❹	loop in main	audio_timer_timeout	radio_nrf5_isr	isr_cb_param
	Misc	button	❹	loop in main	button_pressed	nrfx_gpiote_0_irq_handler	m_cb
	Sensor	ens210 adt7420 ams_iAQcore bmi270 dps310 isl29035 max17262 mpu6050 grove_light grove_temperature dht icm42605					
RIOT	Network	ccn-lite-relay	❷❹	loop in main	_ccnl_content, _ccnl_fib	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		cord_ep	❷❹	loop in main	_cord_ep_handler, _gnrc_netif_config	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		cord_epsim	❷❹	loop in event_loop	handler	isr_rfcorerxtx	cc2538_rf_hal, cc2538_state
		cord_lc	❷❹	loop in main	cord_lc_cli_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gnrc_lorawan	❷❹	loop in main	_gnrc_netif_config, _pm_handler	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		dtls_echo	❹	loop in main	udp_client_cmd, udp_server_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		dtls_wolfssl	❹	loop in main	dtls_client, dtls_server	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		dtls_sock	❷❹	loop in main	dtls_client_cmd, dtls_server_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gcoap_example	❷❹	loop in main	gcoap_cli_cmd, _xfa_6ctx_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gcoap_block_server	❷❹	loop in main	gcoap_cli_cmd, _xfa_6ctx_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gcoap_dtls	❷❹	loop in main	gcoap_cli_cmd, _xfa_6ctx_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gcoap_fileserver	❷❹	loop in main	_xfa_ls_cmd, _xfa_6ctx_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gnrc_border_router	❷❹	loop in main	_xfa_6ctx_cmd, _xfa_ifconfig_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gnrc_networking	❷❹	loop in main	_xfa_6ctx_cmd, _xfa_ifconfig_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gnrc_networking_mac	❷❹	loop in main	udp_cmd, mac_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		gnrc_networking_subnets	❷❹	loop in main	udp_cmd, mac_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		lorawan					
		nanocoap_server	❷❹	loop in nanocoap_server	coap_parse, coap_handle_req	isr_rfcorerxtx	cc2538_rf_hal, cc2538_state
		opendsme_example	❷❹	loop in main	_nimble_netif_handler, _gnrc_netif_config	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		paho_mqtt_example	❷❹	loop in main	_nimble_netif_handler, _gnrc_netif_config	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		spectrum_scanner	❹	loop in spectrum_scanner	print_u32_dec, print_u64_dec, print	isr_rfcorerxtx	cc2538_rf_hal, cc2538_state
		telnet_server	❷❹	loop in main	_gnrc_6ctx, _gnrc_netif_config	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		twr_aloha	❹	loop in main	_twr_handler, _twr_ifconfig	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
		wakaama	❹	loop in main	lwm2m_cli_cmd	isr_rfcorerxtx	isr_ctx, cc2538_rf_hal, cc2538_state
	Storage	filesystem	❷❹	loop in main	_xfa_ls_cmd, _cat, _tee	isr_uart0	isr_ctx
	Misc	lua_basic micropython openthread example_psa_crypto thread_duel timer_periodic_wakeup saul					
Mbed	Bluetooth	BLE_Advertising	❹	loop in equeue_dispatch	start_advertising	radioCback	driverState, tifsState
		BLE_GAP	❹	loop in equeue_dispatch	onConnectionComplete, onDisconnectionComplete	radioCback	driverState, tifsState
		BLE_GattClient_CharacteristicUpdates	❹	loop in equeue_dispatch	start_discovery	radioCback	driverState, tifsState
		BLE_GattClient_CharacteristicWrite	❹	loop in main	onDataRead, onDataWritten	radioCback	driverState, tifsState
		BLE_GattServer_AddService	❹	loop in equeue_dispatch	start_advertising	radioCback	driverState, tifsState
		BLE_GattServer_CharacteristicUpdates	❹	loop in equeue_dispatch	onDataRead, onDataWritten	radioCback	driverState, tifsState
		BLE_GattServer_CharacteristicWrite	❹	loop in equeue_dispatch	onDataWritten	radioCback	driverState, tifsState
		BLE_GattServer_ExperimentalServices	❹	loop in equeue_dispatch	start_advertising	radioCback	driverState, tifsState
		BLE_PeriodicAdvertising	❹	loop in equeue_dispatch	onConnectionComplete	radioCback	driverState, tifsState
		BLE_SecurityAndPrivacy	❹	loop in equeue_dispatch	onConnectionComplete	radioCback	driverState, tifsState
		BLE_SupportedFeatures	❹	loop in equeue_dispatch	on_init_complete	radioCback	driverState, tifsState
	NFC	NFC_EEPROM	❹	loop in equeue_dispatch	on_ndef_message_written	radioCback	driverState, tifsState
		NFC_SmartPoster	❹	loop in equeue_dispatch	on_nfc_initiator_discovered	radioCback	driverState, tifsState
	Network	mbed-os-example-cellular	❹	loop in equeue_dispatch	test_send_and_receive	CAN1_RX0_IRQHandler	can_irq_contexts
RThread	File Parser	json	❶❹	loop in _svfscanf_r	cJSON_Parse	USART1_IRQHandler	uart_obj
		mp3decode	❶❹	loop in dfs_file_read	MP3Decode	USART1_IRQHandler	uart_obj
		nmea	❶❹	loop in _svfscanf_r	nmea_parse	USART1_IRQHandler	uart_obj
		qrcode	❶❹	loop in _svfscanf_r	qrcode_getModule	USART1_IRQHandler	uart_obj
		sms	❶❹	loop in _svfscanf_r	sms_deliver_parse	USART1_IRQHandler	uart_obj
		xml	❶❹	loop in _svfscanf_r	ezxml_parse_str	USART1_IRQHandler	uart_obj
ChibiOS	Misc	NIL-STM32F401RE-NUCLEO64-LCD_II-HD44780	❸	loop in main	palToggleLine, chnWrite	chSysTimerHandlerI	nil
		NIL-STM32F401RE-NUCLEO64-LCD_II-HD44780	❸	loop in main	palToggleLine, lcdBacklightOff, lcdBacklightOn	chSysTimerHandlerI	nil
		RT-STM32F303-DISCOVERY-S3DL	❸	loop in main	s3dl_games_handler	chSysTimerHandlerI	nil
		RT-STM32F401RE-NUCLEO64-ADC-Joystick	❸	loop in main	palTogglePad, chprintf	chSysTimerHandlerI	nil, flag
		RT-STM32F401RE-NUCLEO64-ADC-Slider	❸	loop in main	palTogglePad, chprintf	chSysTimerHandlerI	nil, flag
		RT-STM32F401RE-NUCLEO64-LCD_II-HD44780-PCF8574	❸	loop in main	palReadPad, palClearPad, palSetPad	chSysTimerHandlerI	nil
		RT-STM32F401RE-NUCLEO64-LCD_II-HD44780	❸	loop in main	lcdBacklightOff, lcdBacklightOn, palToggleLine	chSysTimerHandlerI	nil
		RT-STM32F401RE-NUCLEO64-HC-SR04+HD44780-PCF8574	❸	loop in main	palClearPad, palSetPad	chSysTimerHandlerI	nil
Contiki	Network	udp-server	❹	process_run	udp_rx_callback	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		snmp-server	❹	process_run	udp_rx_callback	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		border-router	❹	process_run	httpd_appcall	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		sixp-node	❹	process_run	sixp_output	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		channel-selection-demo	❹	process_run	process_thread_tsch_process	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		lwm2m-ipso-objects	❹	process_run	lwm2m_rd_client_register_with_server	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		multicast	❹	process_run	uip_htonl, printf	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		ip64-router	❹	process_run	ip64_eth_interface_input	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		sensniff	❹	process_run	execute_command	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		nullnet-broadcast	❹	process_run	input_callback	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		coap-example-server	❹	process_run	process_thread_er_example_server	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
		slip-radio	❹	process_run	slip_input_callback	cc2538_rf_rx_tx_isr	cc2538_rf_process, poll_requested
	Misc	antelope-shell	❹	process_run	db_query, db_processing	uart_isr	rxbuf, cc2538_rf_process, poll_requested