Stop Starving or Stuffing Me: Boosting Firmware Fuzzing Efficiency with On-demand Input Delivery

Shandian Shen*, Wei Zhou*✉, Keming Zhao*, Peng Liu†, Chung Hwan Kim‡, Le Guan§
*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
*Hubei Key Laboratory of Distributed System Security
†Penn State University, ‡University of Texas at Dallas, §University of Georgia
E-mails: {shenshandian, weizhou_sec, zhaokeming}@hust.edu.cn, pliu@psu.edu, chungkim@utdallas.edu, leguan@uga.edu

***Abstract*—Firmware fuzzing has gained attention for its ability to identify firmware bugs. While progress has been made in firmware emulation to support fuzzing, current approaches often directly integrate fuzzing tools for general software. However, unlike general software, which receives input as it encounters I/O functions, firmware input can be received asynchronously and independently of the firmware's execution, with uncertain *timing and quantity*. Without full awareness of firmware's exceptions, existing solutions often imprudently deliver fuzzer-generated input to the firmware in an ad-hoc way. This either overwhelms the processing function of the firmware (i.e., *stuffing* problem) or fails to deliver enough input data to trigger input processing functions (i.e., *starving* problem). In both cases, fuzzing capability is weakened.**

In this paper, we comprehensively investigate the input delivery issue, a unique and less studied field in firmware fuzzing. To accurately determine the optimal timing and quantity for delivering test cases, we leverage the fact that firmware has to check input availability before using any data. Therefore, we employ static and dynamic analysis to map each input processing route into three stages: input retrieval, availability check, and processing. This recovered semantic information allows the fuzzer to accurately deliver input at the availability check points within the expected length range. Since firmware may have multiple input routes, we also optimize the scheduling algorithm to reach more diverse routes. Our prototype, named *FIDO*, can serve as an add-on to existing firmware fuzzers to enhance their test-case delivery effectiveness. Compared to ad-hoc input delivery methods used in *Fuzzware* and *MULTIFUZZ*, *FIDO* increases their median code coverage by up to 115% and 54%, respectively. Compared to *SEmu*, which requires humans to manually specify input delivery points, *FIDO* still improves its coverage by up to 19%. As a result of improved input delivery strategy, *FIDO* discovers known bugs significantly faster and also identifies five previously unknown bugs.

The first two authors contributed equally (alphabetical order).
Wei Zhou is the corresponding author.

1. Introduction

Microcontroller Unit (MCU) based embedded devices are widely used in security- and safety-critical sectors such as infrastructure, smart homes, and healthcare. At the same time, vulnerabilities in MCU-based devices have steadily risen. Over the past decade, fuzzing has emerged as a highly effective technique for detecting software vulnerabilities. Therefore, researchers have been working on adopting traditional fuzzers for firmware testing which relies on real hardware [31], [33], [35], [57] or an emulator [16], [48], [52], [56], [59], [60] for code execution. Beyond traditional fuzzing, recent studies also retrofit fuzzing strategies to accommodate firmware-specific features [7], [19], [50]. For example, the authors of *Hoedur* [50] and *MULTIFUZZ* [7] found that firmware receives inputs from multiple peripheral interfaces. They correspondingly propose to maintain a separate input stream for each peripheral and independently mutate them. Such multi-stream fuzzing reorganizes the previously series single-stream fuzzing input into per-peripheral streams that are independently mutated, enabling peripheral-specific and type-aware fuzzing.

Input Delivery Problem in Firmware Fuzzing. In this paper, we demonstrate that being orthogonal to the multi-stream problem, there is another equally critical problem (i.e., the input delivery problem) caused by firmware specificities: solving this problem can significantly boost fuzzing performance.

We now explain the input delivery problem in firmware fuzzing if this specificity is not handled properly. Traditional software uses well-defined POSIX I/O interfaces such as `read(fd, buf, len)` to synchronously receive input from files or keyboards. Conventional fuzzers simply hook into these I/O APIs to deliver inputs to the corresponding `fd`, ensuring inputs are always available without needing to manage delivery quantity, as the program's function parameters dictate the expected input byte amount. In contrast, firmware retrieves and processes input from various peripherals with *non-standard* I/O interfaces and *asynchronous* triggering mechanisms (*e.g.,* interrupt), making the *arrival time and quantity* of peripheral input largely unpredictable.

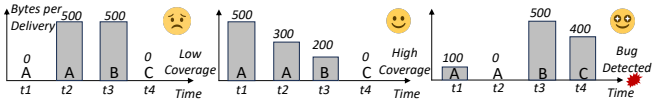


Figure 1: Three delivery plans with varying timing and quantities (A, B, and C refer to different input-taken points).

Without the knowledge of expected input arrival time and quantities, existing fuzzers make ad hoc input delivery decisions/plans and fail to optimally deliver input, often leading to low fuzzing effectiveness and efficiency. We illustrate three possible input delivery plans in Figure 1. In each plan, assuming that in total 1,000 bytes are delivered to three input-taken points A, B, and C. A fuzzer decides whether to deliver input and how many bytes at each time slot *t1* to *t4* when encountering these points. 0 bytes means that the fuzzer determines not to deliver any input at that time slot. Depending on the firmware, these plans have led to very different code coverage and bug detection outcomes. The first plan fails; the second plan increases code coverage; the third plan detects a bug.

Limitations of Existing Works. Existing firmware fuzzers deal with input delivery in ad-hoc ways, leading to sub-optimal efficiency. For instance, *Fuzzware*, *Hoedur*, and *MULTIFUZZ* support two fuzzing modes: periodic and fuzz. In the periodic model, a few bytes are delivered by triggering interrupts at fixed intervals (*e.g.,* every 1,000 basic blocks). The fuzz mode is similar to the periodic mode, except that the interval and interrupt are dynamically determined at runtime based on the fuzzing input. These earlier works, without understanding firmware's expectation, are more likely to inject extra bytes when the firmware does not expected any, or fail to deliver bytes when the firmware really needs data. *SEmu* relies on manually specified delivery points, but does not account for the size limit of peripheral buffers. This causes the old buffer to be overwritten, thus wasting test cases. Additionally, manual configuration is also labor-intensive and may not capture all the valid input delivery points, as we demonstrate in evaluation.

A concurrent work, *AidFuzzer* [54], aims to determine when firmware is awaiting interrupts by identifying waiting states like infinite loops or points reading global variables influenced by interrupt handlers. The heuristic is that when the firmware communicates with the interrupt handler, which handles low-level data reception, it expects new data. If such a state is encountered a specified number of times, *AidFuzzer* triggers an interrupt. However, these waiting states do not necessarily represent the firmware's genuine intention to consume input. As a result, this coarse-grained heuristic often over-approximates the inference problem, flagging cases where no input is actually required. We discuss problems with the coarse-grained heuristic in evaluation (Section 6.3).

Our Solution. Our work makes three key contributions to facilitate more efficient and effective firmware fuzzing. First, we propose a new approach to more precisely identify input retrieval points. Rather than inferring waiting states through accesses to shared variables, our approach identifies the complete control-flow route along which input is Checked (for availability), Retrieved, and subsequently Processed—a sequence (of operations) we term a "**CRP input route**". For each such route, input is delivered directly at the check locations. This semantic-level reasoning provides a more reliable indication that the firmware genuinely requires input. Second, our approach can determine the range of input sizes that the firmware can effectively process. The proposed "*watermark*" technique gradually increases input length until the firmware begins to process input, indicating the lower limit. The upper limit is detected when the delivered input continues growing and leads to the old data being overwritten in the receive buffer. Lastly, we propose a multi-route-aware input delivery strategy. Firmware can have multiple input routes, each with varying input length requirements and occurring under different calling contexts. These input routes may be hit at different times during fuzzing. Without coordinating input delivery for these routes, most inputs would be directed to a small group of routes, leaving others less tested. The proposed strategy dynamically predicts potential input routes and distributes test cases fairly among all input routes.

We have implemented our idea with a prototype named *FIDO* (acronym for Fuzzing Input Delivery Optimizer). We have integrated *FIDO* as an input delivery plugin with three state-of-the-art (SOTA) firmware fuzzers: *Fuzzware* [48], *SEmu* [60] and *MULTIFUZZ* [7], replacing their original input delivery mechanisms. Based on our evaluation of 28 unit tests and 25 real-world firmware images, *FIDO* accurately identified optimal delivery strategies for all samples. With more efficient input delivery, *FIDO* reduces input wastage and increases likelihood of passing availability checks. This improvement translates to significantly boosted fuzzing capability. In our five groups of 24-hour fuzzing campaigns for real-world firmware images, *FIDO* increases median code coverage by up to 115% and 54% compared to the periodic delivery methods of *Fuzzware* and *MULTIFUZZ*, respectively, and by up to 106% compared to *Fuzzware*'s fuzzed mode. Compared to SEmu, which requires humans to manually specify input delivery points, *FIDO* still improves its coverage by up to 19%. *FIDO* enhances fuzzers' bug-finding capabilities, triggering crashes 1 to 100 times more often than these with the original ones, and discovering known bugs hundreds of times faster. With the help of *FIDO*, we found five new bugs and one of them was assigned with CVE number.

In summary, our contributions are four-fold:

- We reveal the test case delivery problem, a firmware-specific roadblock hindering efficient firmware fuzzing.
- We address the test case delivery problem using input route analysis. By abstracting an input route with the proposed CRP model (availability **C**heck, data **R**etrieval, **P**rocessing), we can understand when the firmware really needs input and how many bytes to deliver.
- We implement our idea as a plugin called *FIDO* for four SOTA firmware fuzzers.

- Our evaluation results confirm that *FIDO* benefits all the four SOTA firmware fuzzers, effectively improving test case usage, increasing code coverage, and triggering more crashes within shorter time.

To facilitate further research, we have released the *FIDO* source code and dataset at https://github.com/IoTS-P/FIDO.

2. Background

2.1. MCU Firmware and Peripheral Interface

Firmware is a program dedicated for an embedded device. When running on an MCU-based embedded device, it is often monolithic, comprising application code, drivers, libraries, and an optional real-time kernel altogether. If a real-time kernel is included, the firmware is said to be RTOS-based; otherwise, it is said to be bare-metal. Since MCU firmware is dedicated for a particular task, it works in an infinite loop that continuously receives input data from the external world, processes it, and responds to the external world via actuators when necessary.

MCU firmware lacks a unified machine abstraction layer, requiring direct interaction with peripherals through low-level hardware interfaces. There are three main I/O mechanisms in MCUs. First, Memory-Mapped I/O (MMIO) maps peripheral registers into the system address space, allowing firmware to read or write data registers (DR) for input or output. Second, peripherals can signal status changes via Interrupt Requests (IRQs), managed by an interrupt controller like the nested vector interrupt control (NVIC) in ARM Cortex-M MCUs. Third, peripherals such as USB and Ethernet use Direct Memory Access (DMA) for bulky data transfers between RAM and peripherals without processor intervention, enhancing throughput.

2.2. I/O Operating Modes on MCUs

The firmware retrieves external input from peripherals either via reading data register or DMA transfers. Since peripheral hardware operates asynchronously with the processor core, it must notify the firmware of input arrival before reading it. The notification is achieved either passively by letting the firmware poll the status register (polling mode), or actively by triggering an interrupt to the current execution (IRQ mode). Using two simple unit tests of UART peripheral on STM32F429 as an example, we demonstrate the workflows of these two operating modes in Figure 2.

Polling Mode. When external data reaches a peripheral, it updates the status registers (SR). In the UART example, this corresponds to setting the `RXNE` field of the SR, indicating that the peripheral is ready to be read (①). Before accessing the data register (DR), the firmware must check the `RXNE` field (②). The read operation can continue until the `RXNE` field is cleared (`Empty` state) by the peripheral, which means that the input is read out (③).

IRQ Mode. Polling mode inefficiently uses CPU cycles for input availability checks. This can be mitigated by the interrupt mechanism which allows the hardware to automatically

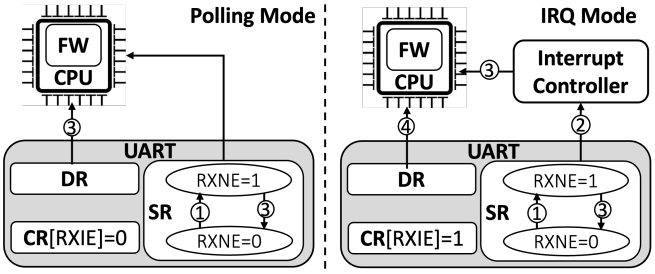


Figure 2: Two I/O operating modes for peripheral input arrival. FW refers to firmware.

notify firmware of data arrival. In the UART example, the firmware can enable interrupt by setting corresponding bit in the `RXIE` field in the control register (CR). When UART input arrives, not only the `RXNE` field in SR is set (①), but also the corresponding peripheral interrupt is activated by the interrupt controller (②). This automatically switches the processor from its current execution context (thread mode in ARM Cortex-M) to a predefined interrupt service routine (ISR) context (handler mode in ARM Cortex-M) (③) to read the data. When the peripheral input is read out, the peripheral turns to the `Empty` state (④). The interrupt mechanism eliminates the need for constant peripheral checks at the cost of increased software complexity to deal with asynchrony (detailed later in Section 2.3).

2.3. Firmware Specificity involved in the Programming Model

Special programming models are employed to accommodate the aforementioned I/O operating modes on MCUs, as recognized in the community [55]. In the polling mode, the firmware constantly checks readiness via reading the SR. Once the condition is met, the firmware retrieves and processes the input in the same execution context. When operating in the IRQ mode, the firmware is free to handle other tasks until it is notified of data arrival asynchronously by the interrupt controller. Data is then typically retrieved in the ISR context promptly and the processing is put off to a later stage outside of the ISR context. This work focuses on fuzzing firmware using the asynchronous IRQ mode because existing firmware fuzzers can already handle firmware in the polling mode that synchronously receives and processes input. In what follows, we summarize four unique characteristics caused by the asynchronous IRQ mode. They directly impact the design of the proposed input delivery strategy.

F1: Limited Reserved Buffer for Input Retrieval. In the IRQ mode, an ISR function is activated upon input arrival, which runs asynchronously with the main firmware function. ISRs are short-lived routines and must exit as soon as possible [30]. Therefore, a global buffer is commonly reserved to allow the ISR to quickly receive the data from peripheral and exit. Later, the main function outside the ISR context can process it as needed. Each peripheral in IRQ mode requires at least one dedicated buffer. The size of the

shared buffer varies depending peripheral usage, generally ranging from several bytes to under 1 KB, due to limited SRAM resources in MCUs. Furthermore, to ensure that the mostly recent bytes are still available in case of data burst, the buffer is typically arranged as a ring data structure. That is, when the buffer reaches its limit, the ISR overwrites the oldest bytes with the newly received data.

F2: Buffer Cleanup at Peripheral Reset. Every time when the peripheral needs to be reset into a predictable and clean state, the associated ISR buffer is cleared. This can occur during device reboot or when the peripheral switches between the receiving and transmitting modes in case of shared receiving and transmitting buffer. Obviously, unprocessed data in the buffer during cleanup will be discarded and cannot contribute to the fuzzing process.

F3: Availability and Length Check. Since the ring buffer is filled by ISR functions asynchronously, the firmware must check if there is new data in the buffer before using it. Additionally, some protocols require a minimal amount of data before they can be processed. The firmware thus also needs to perform a length check of data. Data availability check is implemented in a more principled way when the firmware incorporates an RTOS kernel. Specifically, each input source is assigned with a semaphore and each task monitors a set of semaphores. If no semaphore is available for a task, the task is moved to a sleep list. If a semaphore is released from ISRs, the task waiting on this semaphore wakes up. If no task is ready to run, the system defaults to an idle task.

F4: Multiple Processing Points with Different Calling Contexts. During firmware execution, the input processing functions may be invoked at multiple places in different contexts. The number of consuming points encountered can vary depending on the input data. Additionally, the behavior of the processing function may change according to the calling context.

Generality Study. To assess the generality of these characteristics in real MCU firmware, we examined 87 unit test demos shipped with official SDKs of top MCU vendors as listed in Appendix A, including STM32, NXP, and Microchip, and popular MCU driver library Arduino as well as leading RTOSs like FreeRTOS, Zephyr, Mbed OS, RIoT, and Nuttx, covering commonly-used peripherals such as GPIO, I2C, UART, ADC, SPI. Among the 87 unit test samples, 54 incorporate F1. The remaining samples that do not implement F1 primarily utilize very simple peripherals for direct data transfer, thereby eliminating the need for a ring buffer in RAM. Exceptions include certain samples involving GPIO, and ADC. Due to the inherently asynchronous interaction between peripherals and firmware, all samples incorporate F3. No unit test was found to implement F4, as these tests are specifically designed to validate minimal, isolated functionalities of individual peripherals. In contrast, our analysis of the real-world firmware dataset shows that 19 out of 25 samples include F4.

2.4. CRP Modeling for MCU Firmware

According to the programming model, we found that the operational process firmware uses to handle an input consists of three steps: *availability check, retrieval, and processing (CRP).* For clarity, we define the CRP operations in this work as follows:

- **Availability Check (C_A).** In the polling mode, the firmware directly checks if the SR field which indicates readiness (e.g., the `RXNE` field in UART SR is set in section 2). In the IRQ mode, the main firmware logic checks the buffer populated by the ISRs. In RTOS-based firmware, the semaphore of the input source is checked. In addition, some firmware also performs a minimum length check on the accumulated data in the buffer, denoted as C_L.
- **Data Retrieval (R).** In the polling mode, after confirming the SR field, the firmware reads the DR as data retrieval. In the IRQ mode, data retrieval occurs at two levels. First, the ISR reads from the peripheral's data I/O and stores it in the buffer (low-level data retrieval, R_P). Second, after the low-level availability check, the main firmware logic retrieves data from the ISR buffer (high-level data retrieval, R_B). If R_B occurs during buffer cleanup (F2), it is denoted as $\underline{R_B}$.
- **Processing (P).** In both polling and IRQ modes, the retrieved data is processed by the main logic of the firmware. Operations involving the input, particularly those influence the branch targets at conditional jumps, can contribute substantially to code coverage. These operations drive the main application logic and form the core of the firmware code.

In IRQ mode, an input handling process creates a distinct sub-tree in the control flow graph, connecting instructions of C_A, R_B, and processing (P), referred to as an input route in this paper.

3. Motivation

3.1. Input Delivery Problem

As with general software, the firmware fuzzing loop involves five modules: test case generation, test case delivery, execution with the test case, feedback collection, and feedback-guided input mutation. In this work, we refer to the firmware executor as the fuzzer back-end, and the test case generation and mutation module as the fuzzer front-end. Due to architectural differences, rehosting has been widely adopted as the fuzzer back-end (*e.g., μEmu* [59], *SEmu* [60] and *Fuzzware* [48]) in firmware fuzzing, where the firmware instructions are translated into host instructions. Regarding the fuzzer front-end, previous works typically adopt test case generation algorithms shipped with AFL/AFL++ or libfuzzer.

Recently, researchers [7], [19], [50] found that firmware exposes many specificities that are incompatible with existing fuzzing front-ends. For example, multi-stream works

TABLE 1: Front-end, back-end and input delivery method supported by existing firmware fuzzers for IRQ mode. (with default configurations in bold)

Firmware Fuzzer	Front-end (Fuzzer)	Back-end (Executor)	Delivery Timing	Delivery Quantity Each Time
P²IM [21]	AFL	QEMU	**RR**	Single ISR Read
μEmu [59]	AFL	S2E	**RR**, MSP	Single ISR Read
Ember-IO [18]	AFL++	QEMU	**RR**	Single ISR Read
Fuzzware [48]	AFL, AFL++	Unicorn	**RR**, Fuzz, MSP	Single ISR Read
SEmu [60]	AFL, AFL++	Unicorn	**MSP**	All Fuzzing Input
Hoedur [50]	LibFuzzer*	QEMU	**RR**, Fuzz, MSP	Single ISR Read
MultiFuzz [7]	AFL*	Icicle [5]	**RR**,Fuzz	Single ISR Read
μAFL [31]	AFL	Hardware	**MSP**	All Fuzzing Input

*: Support multi-stream test case generation and mutation.

such as *MULTIFUZZ* and *Hoedur* enhance input generation and mutation to handle the multi-stream nature of firmware inputs from various peripherals. However, when it comes to input delivery (i.e., when and how many bytes to deliver), we found that existing firmware fuzzers overlook the unique programming model in handling inputs in the IRQ mode, and instead use ad-hoc mechanisms, as summarized in Table 1. For instance, most emulation-based firmware fuzzers use a round-robin injection method (**RR**), delivering input by triggering an ISR after every fixed number (*e.g.,* 1,000) of basic blocks executed. The input quantity per trigger is usually between 1B and 4B, as requested in each ISR. Additionally, *Fuzzware*, *Hoedur* and *MULTIFUZZ* supports a fuzzing-guided delivery method (**Fuzz**), where input is delivered with a single ISR trigger but at different intervals determined by the fuzzing input. If multiple peripheral interrupts are enabled, the interrupt chosen is cycled in RR mode and is determined by fuzzing input in Fuzz mode. In comparison, high-fidelity emulation-based fuzzers like *SEmu* and on-device fuzzers like *μAFL* deliver all fuzzing input at manually specified points (**MSP**), typically at the start of the main loop.

3.2. How Ad-hoc Solutions Fall Short

In this section, we analyze how existing delivery methods *stuff* (problems P1, P2) and *starve* (problems P3 and P4) firmware execution, leading to inefficient fuzzing with real-world firmware (see Listing 1). This firmware operates on a `Heat_Press` device, which receives remote commands via the Modbus protocol over a UART peripheral in IRQ mode and exhibits all the four features mentioned in Section 2.3.

```
1 //ISR Context
2 void UART_IRQHandler() {
3     store_char(&Serial, Serial->UART_DR);
4 }
5 void store_char(RingBuffer *this, char c) {
6     if ((this->Head + 1) & 127 != this->Tail) {
7         this->rx_buffer[this->Head] = c; //R_P
8         this->Head = (this->Head + 1) & 127;
9     }
10 }
11 // Main Execution Context
12 void main(...) {
13   ...
14   Modbus::begin(&slave, ...);
15   while (1) {loop();}
16 }
17 void Modbus::begin(Modbus *const this, ...) {
18     ...
19     while (this->Head - this->Tail != 0)
20         this->port->read();
21 }
22 int UARTClass::read(UARTClass *this) {
23     result = this->rx_buffer[this->Tail];//R_B
24     this->Tail = (this->Tail + 1) & 127;
25     return result;
26 }
27 void loop(...) {
28     Modbus::query(&master, x);
29     switch(slave.state) {
30         case 1: Modbus.poll();
31             if (master.state == 0) {
32                 ...//P
33                 slave.state++;
34                 master.state = 1;
35             }
36         case 2: Modbus.poll();
37             if (master.state == 0) {
38               ...//P
39 }
40 void Modbus::query(Modbus *const this, ...) {
41     while (this->Head - this->Tail != 0)
42         this->port->read();
43     ... //data transmission
44 }
45 int Modbus::poll(Modbus *const this, ...) {
46     if (this->port->available() == 0)//C_A
47         return 0;
48     if (Modbus::getRxBuffer(this) <= 7)//C_L
49         return 1;
50     if (au8Buffer[1] & 0x80)//P
51     ....
52     master.state = 0;
53 }
54 int UARTClass::available(UARTClass *this) {
55     return this->Head - this->Tail & 127;
56 }
57 int Modbus::getRxBuffer(UARTClass *this) {
58     BufferSize = 0;
59     while (this->port->available()) {//C_A
60         auBuffer[BufferSize] = this->port->read();
61         BufferSize++;
62     }
63     return BufferSize;
64 }
```

Listing 1: A simplified code snippet of the Heat Press firmware is provided, with key operations commented in the corresponding lines.

P1: Input Overwritten with Overfeeding. As F1 states, peripheral inputs are asynchronously retrieved into a global buffer by the ISR, and later processed by the main function. If an excessive number of bytes are delivered to the buffer and the main function fails to process them promptly, the buffer may become full, leading to either the loss of incoming data or the overwriting of previously stored bytes. In the example, the ISR function calls `store_char` to read a byte from the DR and store it in `rx_buffer`. If more data remains, the ISR is triggered again, accumulating data in the `rx_buffer`. If input exceeds 127 bytes, the `Head` pointer is reset to zero (Line 8 in Listing 1).

This overfeeding issue is common in fuzzers using the MSP delivery method, which delivers all input at once. Firmware fuzzers, however, do not know the buffer's maximum limit and can generate extremely long data in mutation, especially in the havoc process. Thus, for this example, when fuzzer-generated test case length over 127 the out-length data will overwritten the before one. This issue also occurs with RR and Fuzz delivery methods during rapid

TABLE 2: Comparison testing under different delivery configurations. (The bold line indicates the default bytes per delivery used by *Fuzzware*.)

Delivery Method		# BB Coverage			Vol. Avg.		Time	Problem
Timing	Bytes per Delivery	Min.	Max.	Avg.	Retriev.	Proc.	Avg.(s)	
RR	**1**	**486**	**498**	**491**	**1000**	**861**	**15.66**	**P2-4**
	Rand(0-1000)	410	422	418	245	86	0.71	P1-4
	Rand(7-128)	455	461	457	982	724	0.73	P2-4
Fuzz	**1**	**472**	**502**	**485**	**1000**	**887**	**5.67**	**P2-4**
	Rand(0-1000)	420	438	429	369	102	0.61	P1-4
	Rand(7-128)	457	461	459	984	652	0.65	P2-4
MSP (loop)	Rand(0-1000)	457	504	483	1000	1000	0.79	P1,3
Ideal Placement	**Rand(7-128)**	**493**	**516**	**503**	**1000**	**1000**	**0.68**	**-**

deliveries when R_B is infrequent, but the ISR trigger interval is short. In both cases, the high-level consumer (R_B and P) fails to poll all the received data by R_P in time causing input wastage. Overfeeding not only wastes fuzzing input but also causes the fuzzer to spend significant time mutating unused inputs.

P2: Input Discard Due to Incorrect Timing. As mentioned in F2, data retrieved via DR is discarded during buffer cleaning ($\widetilde{R_B}$). In the example, the `Modbus.begin` function clears the current UART input buffer by retrieving all data without using or storing it at Line 20. Similarly, `Modbus.query` retrieves buffer data before transmission.

The firmware fuzzer cannot differentiate between data from $\widetilde{R_B}$ and normal R_B as they use the same instruction. If RR or Fuzz mode provides input bytes before `Modbus.begin` or `Modbus.query`, these bytes are wasted, leading the real processed volume of input can be smaller than the volume retrieved. A similar issue can arise with MSP when the specified delivery points before $\widetilde{R_B}$. Identifying all $\widetilde{R_B}$ requires significant manual effort, as it can occur under specific conditions in the firmware logic, as seen in *SEmu* configurations where delivery happens after the `Modbus.begin` function but the delivery point is incorrectly set before the `query` function.

P3: Unnecessary Availability & Length Check. If no available data to main input source, firmware would repeat availability checking, wasting execution time. In the example, the `main` function runs an infinite `loop()` that calls the `Poll` function. This function uses `available` to check input availability by comparing the buffer's head and tail pointers. In addition to availability, Line 48 also ensures that the data length meets the Modbus protocol's minimum requirement (>7), before processing the data in `auBuffer`. The RR and Fuzz methods do not specify delivery timing, making it difficult to timely delivery before availability checks. This limitation becomes particularly pronounced in scenarios where only few bytes is transmitted as single ISR trigger (which is default configuration) per delivery and additional length check is needed.

P4: Difficulty in Exploring New Behaviors. As described in F4, the firmware retrieves data from different program points at various times. If the fuzzer fails to deliver input at certain points, some availability checks may fail, missing processing functions. In this firmware, although the `Poll` function only retrieves data from UART, the `Poll` function is called in multiple switch-case branches under different contexts. If all the input are delivered to a fewer cases, some cases will have no data to retrieve, causing certain processes to be skipped. RR and Fuzz is hard to delivery the input in time for all these case in random pattern. For the MSP, delivering data only once at beginning of `loop` execution ensures that the `Poll` function has input at Line 30, but leaves no data for the remaining `Poll` invocations (*e.g.,* Line 36). Consequently, the processing code starting at Line 38 is not executed.

Key Idea of Optimal Delivery. To address the identified issues, the optimal delivery time and quantity should meet the following requirements:

- Input delivery timing should occur only at the initial input availability checkpoint of each input route. This prevents issues P2 (where data retrieved by $\widetilde{R_B}$ cannot be processed and included in an input route), P3, and P4.
- The length of delivered input bytes should remain below the ISR buffer's maximum capacity to avoid P1.
- Delivery input bytes must satisfy the minimum length requirement and ensure that input exceeds this length with each delivery to prevent P3.

Delivery Strategy Comparison Demo. To verify our idea and assess how different input delivery strategies influence fuzzing progress, we fuzzed the firmware in Listing 1 using *Fuzzware*, which supports RR, Fuzz and MSP delivery strategies. We extended it to include waiting state monitoring for the wait delivery strategy. The demo runs each strategy using an identical batch of five 1,000B random inputs. This simulates a typical firmware fuzzing process in which the firmware runs indefinitely until all the input bytes are consumed with injected interrupts. The demo also runs a manually specified ideal configuration according to our idea, where inputs are delivered at the first `available` function invocation in the `Poll` function. Delivery quantity varied among the four strategies. The ideal strategy delivered 7 to 128 bytes each time to meet firmware requirements. Detailed configurations are in Appendix B.

We compared average time consumption, total retrieved data (via `rx_buffer`), total processed data (bytes filled into `auBuffer`), and code coverage. As shown in Table 2, the ideal strategy (with manual effort) achieved the highest code coverage, no input wastage, and nearly the shortest time. The other tested strategies cannot 1) guarantee that the retrieved and processed data volume matches the delivery volume, 2) limit execution time, or 3) cover all input retrieval/processing points simultaneously. This work aims to achieve similar results as this ideal configuration via automated program analysis.

4. Design

4.1. Objective and Threat Model

We seek to enhance firmware fuzzing front-end by optimizing the timing and quantity of test case delivery,

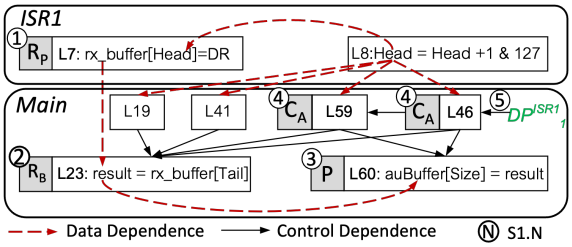


Figure 3: CRP input route mapping process with example in Listing 1. (L#: Line Number in Listing 1)

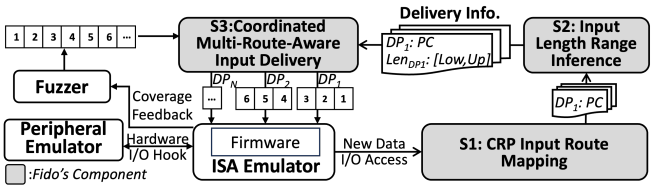


Figure 4: Overview of *FIDO*

without altering other components of the existing firmware front and back-end. This is specifically for firmware that receives external data inputs through IRQ mode. We exclude hardware bugs related to erroneous hardware responses, such as status or control register manipulations, and focus on improving the firmware fuzzer to detect memory-related bugs triggered by external input data, aligning with existing firmware fuzzers.

4.2. Challenges and High-level Insight

The diversity and customization of input handing in real firmware implementations pose the following challenges to extract the semantic information for optimal input delivery.

Accurate Availability Check Identification: While availability checks are performed on global objects modified by ISRs (termed as *availability check variables*), not all these objects are used for such checks. Moreover, global objects used for availability checks may also serve other checks (*e.g.,* `Head` pointer will also be checked for $\widetilde{R_B}$ operations as shown in Line 20 in Listing 1), but should not be used for input delivery (avoiding P2). Lastly, availability checks may occur multiple times before P operations; repeated delivery before the same P can lead to the P1 and P3 problems.

Insight: The CRP operations within the same input route exhibit tight data and control dependencies, as illustrated in Figure 3. Accordingly, we begin by hooking the data I/O access (R_P), then map and connect other CRP operations to form input routes, allowing us to accurately identify the initial availability check for each input route.

Implicit Length Requirement. Since input data transfers through multiple buffer objects before processing, identifying the accumulated length in each buffer for length checks is challenging. Tracking the variables indicating length is also tedious. Additionally, the ISR buffer's maximum capacity is determined by firmware logic and differs from the allocated memory space, making static analysis of ISR buffer objects unreliable. Symbol and debug information should not be relied upon as they typically do not exist in stripped firmware.

Insight: Instead of relying on specificity-ignoring applications of existing dynamic and static analysis techniques, we can leverage the inherent characteristics of the input handling programming model to infer the length range: 1) As noted in F3, P operations occur only when length checks (*i.e.,* lower limits) are satisfied; 2) As noted in F1, the ISR uses a ring buffer for input storage, meaning the input is overwritten before retrieval, indicating the input byte length has exceeded the upper limits. Thus, we propose incrementally increasing the delivery bytes at the delivery point and monitoring the length at which these behaviors occur.

4.3. Approach Overview

Figure 4 illustrates the work flow of our solution, *FIDO*, which is activated during firmware fuzzing when peripheral data I/O access occurs. *FIDO* first maps the program counter (PC) for each CRP operation in the current input route to pinpoint their delivery points (see Section 4.4). Next, *FIDO* determines delivery length boundaries by adjusting the input length at these points (see Section 4.5). The delivery point and its length boundaries constitute the optimal delivery information.

To manage delivery across multiple routes with varying calling contexts, *FIDO* functions as an input delivery extension between the firmware fuzzer's front-end and back-end. In the front-end, it slices and distributes the original test case to each delivery point based on the delivery information (see Section 4.6). In the back-end, it hooks the PC address of each delivery point. When the firmware hits a delivery point, *FIDO* delivers a fuzzing input slice by triggering the corresponding interrupts.

4.4. S1: CRP Input Route Mapping

In this subsection, we detail the sub-steps for the automatic identification of CRP operations and input routes, as illustrated in Figure 3.

S1.1: R_P Identification. During fuzz testing, we hook the MMIO access. If the MMIO access pattern aligns with the DR (as defined in P^2IM [21]) and the CPU is in handler mode, we confirm the data I/O access (R_P) is in IRQ mode (*e.g.,* Line 7). Alternatively, more precise DR address information can be sourced directly from public MCU reference manuals, as shown by *SEmu* [60].

S1.2: R_B Identification. As shown in Figure 3, the R_P and R_B operations are executed in separate firmware contexts. They have no direct control dependence but are directly data-dependent through the ISR buffer (*e.g.,* `rx_buffer`). As noted in F1, since ISR buffers are global and receive

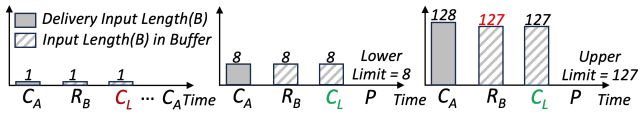


Figure 5: Length range inference with example in Listing 1.

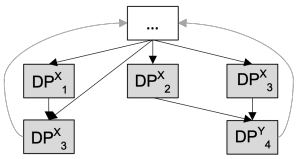


Figure 6: The $\sum^{n} \left|CFG(DP_n^{ISR})\right|$ = 7 in this figure

update only from data registers, we dynamically trace the data flow from the R_P operation to determine the final memory address upon ISR exit, identifying the address of the ISR buffer. By hooking this address, we capture the PC of retrieved buffer data as the R_B operations (*e.g.,* Line 23) when the firmware execution reaches the hook.

S1.3: P Identification. As shown in Figure 3, the processing instruction directly depends on R_B. We use taint analysis from R_B to identify the P instructions, setting R_B as the source and the P instructions, which are conditional branch instructions involving input (*e.g.,* `CMP input, R0`), as the sink. Note that we only need to identify the nearest P instructions in each input route to distinguish between different input route and pinpoint the availability check operation, as detailed later. In Listing 1, the `getRxBuffer` function reads input from `rx_buffer` and compares it with a specific number at Line 50, which is a processing instruction.

S1.4: C_A Identification. For bare-metal firmware, availability checks often rely on a pointer (*i.e., availability check variable*) indicating the storage location of retrieved data. In RTOS-based firmware, a global semaphore (*i.e., availability check variable*) is also modified after R_P. During S1.2, we monitor the global memory reading variable, which is written with different values after R_P as *availability check variables*. These variables are recorded and mapped with corresponding R_P, R_B, and P instructions. We use cross-references of these variables to find conditional branch instructions dependent on them. Note that *availability check variables* may be checked before $\widetilde{R_B}$ for other purposes. We analyze the control dependence between the check and P, considering only checks with reverse control dependence on P and R_B operations as real availability checks. As shown in Figure 3, we identify the `Head` pointer modified in the ISR function. We find checks in Line 19, Line 41, Line 46, and Line 59 where branch instructions invoke the `Head` pointer as shown in Figure 3. Only Line 46 and Line 59, which have reverse control dependency on the P and R_B instructions, are identified as availability checks (C_A). Finally, C_A, R_B, and P are mapped and connected as a sub-graph in the CFG (*i.e.,* an input route).

S1.5: Delivery Point Identification. Multiple availability check points can dominate the same data retrieval point (*e.g.,* Line 46 and Line 59) within an input route. To address this, we analyze the control dependencies among these checks and the control flow. We identify the delivery point that has control dependence over others as the initial availability check for each input route. For instance, since the Line 46 precedes and controls Line 59, we designate Line 46 as a delivery point, mapping it to the corresponding `UART_IRQHandler` ISR function, donated as DP_1^{UART}.

4.5. S2: Input Length Range Inference

We propose a watermark-based method to dynamically determine the limits for input size. Our method gradually increases the length of the tentatively delivered input through additional interrupts with dummy input at the identified delivery point, as shown in Figure 5. Note that this process needs extra dummy-input but runs only once for each delivery point.

Lower Limit Inference. We dynamically hook and monitor under what input length the P instructions (identified in S1.3) are reached. In the `Heat_Press` firmware in Listing 1, we start input delivery with triggering UART interrupt at the delivery point (Line 46) and add code hook at P instruction (Line 32). We gradually increase the delivery length by triggering ISRs (`UART_IRQHandler`) additional times, each adding one byte to the buffer. Then, we found delivering eight bytes enables the firmware to reach P instructions, setting the lower limit.

Upper Limit Inference. We monitor the data written to the ring buffer during ISR. Once any old data is overwritten, we note the length and use that as the buffer's upper limit. In `Heat_Press`, we place a memory hook at the beginning of `rx_buffer` (identified in S1.2) and increase delivery length by triggering additional ISRs, pausing the processing function to accumulate input in the buffer. When inputs exceed 127 bytes, new data would overwrite the old ones, exposing the upper limit of the buffer (on the right of Figure 5).

4.6. S3: Coordinated Multi-Route-Aware Delivery

Firmware can have multiple input routes corresponding to multiple delivery points ($p \in DP$) and each with specific length range requirements ($[LOW_p, UP_p]$). Different fuzzing rounds may encounter different points at various calling contexts, as mentioned in F4. Whenever one DP is being handled, one input route is being tested by the fuzzer. In order to test multiple input routes, we propose a coordinated multi-route-aware input delivery algorithm Algorithm 1, which intends to partition a given test case (FI) into multiple segments: each segment serves one route.

We begin by counting N, the total number of already-identified delivery points with distinct calling contexts (different in-edges of the delivery point in the loop CFG), as

demonstrated in Figure 5. To ensure the fuzzing input can be split into at least N parts (one for each delivery point), each with minimum size X (the smallest lower limit among the current delivery points), we conduct the following steps:

1) Line 4-8: Calculate $Len_R = N \times X$, the total length supposed to be reserved for N minimum-sized chunks. If the length of the fuzzing input is shorter than the supposed reservation, manage the trade-off by reducing N until the length of the fuzzing input is no longer shorter. Note that the trade-off here compromises on delivery point coverage.
2) Line 9: Subtract this reserved length from the total input length. Now, Len_{FI} is the remaining bytes available beyond the reserved chunks.
3) Line 16-20: For assigning the next SI (segment of the input) at the current point, the algorithm first checks if Len_{FI} plus X meets the current delivery point's minimum length.
4) Line 21-26: If it does, the algorithm calculates the delivery length (len_{SI}) of SI as a random value within current delivery point's limits, extracts (the next) Len_{SI} not-yet-used bytes from FI, and delivers SI. To maintain fuzzing consistency (*i.e.,* the same fuzzer test-case always produces the same execution results), the algorithm records the delivered SI as the random seed, ensuring reproducibility.
5) Line 27-32: After allocating SI, update the trackers Len_{FI}, Len_R and Pos. Note that Pos ensures that no already-used byte in FI will be reused.
6) Line 10-15: When $Len_R + Len_{FI}$ is 0, all bytes in FI are delivered, ending the current fuzzing round.

In addition, information of DPs is being updated with the `DPIdentify` function (Steps S1-S2) when new delivery points are identified (Line 12).

Algorithm 1 Coordinated Multi-Route-Aware Delivery

Input: Entry Point (EP), Fuzzing Input(FI), Delivery Point (DP)

1: $Pos = 0$, $PC = EP$
2: $N = \sum^{n} |CFG(DP_n)|$
3: $X = \min_{p \in DP} LOW_p$
4: $Len_R = N * X$
5: **while** $Len_{FI} < Len_R$ **do**
6: $N = N - 1$;
7: $Len_R = N * X$;
8: **end while**
9: $Len_{FI} -= Len_R$
10: **while** true **do**
11: $PC = Execute(PC)$;
12: **if** $PC == p \in DP$ **then**
13: **if** $Len_{FI} + Len_R == 0$ **then**
14: **return**;
15: **else**
16: $\Delta = Len_{FI} + X - LOW_p$;
17: **if** $\Delta \leq 0$ **then**
18: $Len_{SI} = Len_{FI} + X$;
19: $SI = FI_{[Pos, Pos+Len_{FI}+X]} \parallel \underbrace{00\ldots0}_{|\Delta|}$;
20: $Delivery(SI)$;
21: **else**
22: $t = \min(Len_{FI} + X, UP_p) - LOW_p$;
23: $Len_{SI} = Rand(FI_{[Pos]}) \mod t + LOW_p$;
24: $SI = FI_{[Pos, Pos+Len_{SI}]}$;
25: $Delivery(SI)$;
26: **end if**
27: $Len_{FI} -= Len_{SI} - X$;
28: $Len_R -= X$;
29: $Pos = Pos + Len_{SI}$;
30: **if** $Len_R == 0$ **then**
31: $X = 0$;
32: **end if**
33: **end if**
34: **end if**
35: **if** $PC == New\ R_P\ or\ R_B$ **then**
36: $DP += DPIdentify(PC)$
37: **end if**
38: **end while**

5. Implementation

We implemented our prototype, *FIDO*, as a plugin for three state-of-the-art firmware fuzzers: *MULTIFUZZ*, *SEmu*, and *Fuzzware*. These fuzzers originally employ various delivery strategies as listed in Table 1 with single- or multi-stream support. This plugin implementation demonstrates *FIDO*'s compatibility and allows for a comprehensive evaluation of different delivery strategies.

FIDO consists of a set of static and dynamic analysis modules that interact via a Ghidra server [2]. The static analysis module employs Ghidra [1] for the sub-steps of S1. In step S1.3, *FIDO* leverages Ghidra's `getDescendants` for data flow taint analysis to find the nearest compare branch instruction (PCode type `INT_EQUAL`) as P instructions. In step S1.4, it uses Ghidra's `getReferencesTo` API to collect reference information for global variables. Step S1.5 involves using Ghidra's `getSources` API to backtrack the control flow of each basic block to identify initial availability check instructions.

The dynamic analysis for steps S1, S2, and S3 is customized for each emulator back-end with a similar interface: *Fuzzware* and *SEmu* utilize Unicorn, while *MULTIFUZZ* is based on Icicle [5]. We use Unicorn as an example to illustrate our implementation. The same idea applies to Icicle, albeit with different APIs. For Unicorn, in step S1.1, we use the `UC_HOOK_MEM_READ` API for MMIO access monitoring to identify R_P. In step S1.2, we use `UC_HOOK_MEM_WRITE` API to track the global ISR buffer access and identify modified global variables. Static analysis in steps S1.3, S1.4, and S1.5 may struggle with indirect jumps, which we address using a hybrid method similar to recent works [37], [61] to solve them on demand. In step S2, `UC_HOOK_CODE` API at each P instruction determines the lower limit, while `UC_HOOK_MEM_WRITE` API at the beginning memory address of ISR buffer determines the upper limit. We note that dynamic hooks in S1 and S2 are a one-time setup and are removed once delivery points and lengths are determined. In step S3, we use the `UC_HOOK_CODE` API to hook all delivery points (DPs) to manage the delivery of fuzzing input generated by the fuzzer front-end, as outlined in Algorithm 1. Specifically, we trigger the corresponding interrupt for SI delivery at specific times at the DP Hook.

6. Evaluation

We evaluated *FIDO* to answer the following research questions (**RQs**):

RQ1: Can *FIDO* automatically identify input routes in both unit tests and real-world firmware? (Section 6.1)

RQ2: By addressing the delivery issues outlined in Section 3.2, to what extent does *FIDO* improve code coverage and bug-finding capability compared to the ad hoc delivery strategies used by SOTA firmware fuzzers? (Section 6.2)

RQ3: How does *FIDO* compare with other interrupt-driven firmware fuzzers, such as *AidFuzzer*, in addressing the identified delivery issues? (Section 6.3)

RQ4: To what extent does each component (*i.e.,* S1, S2, and S3) contribute to fuzzing effectiveness? (Section 6.4)

Firmware Samples Collection. We selected 28 unit test samples from our large-scale empirical study in Section 2.3. These samples include driver code for common data I/O peripherals (GPIO, UART, I2C, ADC), HALs from top MCU vendors (STM32, NXP, Arduino), and RTOS kernels (RIoT, Nuttx). We tested 25 real-world firmware samples, including 22 that were tested by *SEmu*, *Fuzzware*, *AidFuzzer* and *MULTIFUZZ*, and have at least one data peripheral in IRQ mode. We excluded samples without peripherals operating in IRQ mode. We also excluded samples that only timer peripherals in interrupt mode, as *FIDO* does not contribute to them. We also excluded DMA firmware and `STM_PLC` samples, as neither *AidFuzzer* nor *FIDO* supports DMA emulation and nested interrupt triggering. We also incorporated two BLE GATT server examples from the MbedOS BLE project [36] (`Gatt_Clientupdate` and `Gatt_Serverupdate`) and one `BLE_HCI` example from the Zephyr project [58]. Details about these samples, including MCU model, OS/library, and total basic block number, are detailed in Table 10 in Appendix E.

Experiment Setup. All experiments were conducted on a PC equipped with an Intel Xeon Platinum 8350C processor at 2.60GHz, 256GB RAM, and a 960GB SSD storage.

6.1. Delivery Information Identification (RQ1)

A unit-test samples usually tests a single peripheral in an infinite main loop. We manually verify the delivery information identified by *FIDO* once a main loop completes. In contrast, real-world firmware often contains multiple input routes with varying activation conditions. To cover as many input routes as possible, we collect the resulting delivery information after 24 hours of fuzzing and then manually verify its accuracy.

FIDO accurately identifies delivery information of each input route with details listed in Appendix C and Appendix E. Firmware varies in the number of input routes, peripherals, and length requirements. Within a firmware sample, some input routes consistently serve as main inputs during fuzzing, while others appear only under specific conditions. This variability highlights the challenges in automatically and accurately identifying the different stages within an input route.

Time Usage. We measure the time from hitting the retrieval hooks to extracting the delivery information. Identifying one delivery point (S1) takes 3–10 seconds (8.84 seconds on average), while length inference (S2) takes 2–7 seconds (5.06 seconds on average). Overall, extracting delivery information per sample takes less than 20 seconds, highlighting the efficiency of our approach. Additionally, for an input route, delivery information extraction is a one-time effort.

6.2. Fuzzing Efficiency Improvement (RQ2)

For RQ2, we compare the original delivery methods (RR and Fuzz modes for both interrupt selection and interval) shipped with SOTA firmware fuzzer (*i.e., Fuzzware*, *MULTIFUZZ* and *SEmu*), against the *FIDO* method integrated with the same fuzzing tools. For each target, we conducted five iterations using different random seeds.

The fuzzing improvement potential of *FIDO* depends on the extent to which the existing ad-hoc delivery strategies suffer from problems P1-P4. Addressing P1 and P2 maximizes fuzzing input capacity, solving P3 allows the executor to focus on exploring input processing code spaces where real exploitable bugs exist faster, and resolving P4 enables fuzzing to achieve greater code coverage. For the samples *i.e.,* `6LoWPAN_Receiver`, `6LoWPAN_Transmitter`, `P2IM_Drone`, *FIDO* shows minimal performance improvement, as detailed in Appendix E. A manual check reveals that the fuzzer rarely triggered data peripherals in interrupt mode for these samples. For instance, the UART peripheral of `P2IM_Drone` in interrupt mode is only used during the firmware setup process.

6.2.1. Code Coverage Improvement. As shown in Table 3, *FIDO* achieves higher median coverage across most samples than *MULTIFUZZ* and *Fuzzware*, regardless of RR or fuzz mode, particularly for complex firmware with multiple input routes (F4) like `Gateway`, `LiteOS_IoT`, `3D_printer`, and `CCN-Lite-Relay`.

This is because RR or fuzz mode configurations struggle to deliver inputs promptly, particularly for routes that are activated only under specific conditions. For instance, the `Gateway` firmware primarily uses UART for continuous command processing, while other routes are only activated under specific command value via UART. When the command value equals `0x78`, the I2C peripheral can be enabled, hitting the I2C availability check. If the check fails, this input route cannot be accessed again in the same fuzzing round. In both RR and Fuzz modes, delivering input at this precise moment is extremely difficult. By hooking the I2C availability check operation, we found that in *MULTIFUZZ*'s RR delivery method, although the I2C availability check was reached 790,060 times, it failed to trigger the I2C interrupt for input delivery, missing coverage of I2C input processing.

Addressing P2-P3 accelerates coverage exploration. As also shown in Figure 8 in Appendix F, *FIDO* increases coverage faster than the RR or fuzz strategy. Note that in RR or

TABLE 3: Code coverage (median of 5 trials after 24-hours) using *FIDO* compared to original delivery methods. Shaded areas means indicating additional coverage from bug exploits. Growth Rates (GRs) that have significant changes are marked in bold (based on a Mann-Whitney U test with a 0.05 significance threshold).

Firmware	Feature	*Fuzzware*							*MultiFuzz*						
		RR	Fuzz	w.*FIDO*	Problem	GR(RR)	Problem	GR(Fuzz)	RR	Fuzz	w.*FIDO*	Problem	GR(RR)	Problem	GR(Fuzz)
3DPrinter	F1,F2,F3,F4	786	780	**931**	P2-P4	**+18.4%**	P2-P4	**+19.3%**	4,193	3,642	**4,411**	P2-P4	+5.2%	P2-P4	**+21.1%**
Bootstrap(SPI)	F1,F3,F4	956	950	**998**	P3	+4.4%	P3	+5.1%	982	1,184	**1,198**	P3	**+22.0%**	P3	+1.2%
Bootstrap(UART)	F1,F3,F4	994	951	**1,878**	P1,P3-P4	**+88.9%**	P1,P3-P4	**+97.5%**	1,289	1,986	**1,986**	P1,P3-P4	**+54.1%**	P1,P3-P4	+0.0%
CCN-Lite-Relay	F1,F3,F4	491	556	**1,054**	P3-P4	**+114.7%**	P3-P4	**+89.6%**	4,077	4,445	**4,472**	P3-P4	**+9.7%**	P3	+0.6%
μtasker_USB	F1,F3,F4	1,269	1,253	**1,518**	P2-P4	**+19.6%**	P2-P3	**+21.1%**	1,995	1,924	**2,129**	P2-P4	+6.7%	P2-P3	+10.7%
Console	F1,F2,F3	711	712	**794**	P2-P3	+11.7%	P2-P3	+11.5%	1,165	1,161	**1,171**	P2-P3	+0.5%	P2-P3	**+0.9%**
Echo_Server	F3	2,854	2,852	**2,905**	P3	+1.8%	P3	+1.9%	3,553	3,567	**3,569**	P3	+0.5%	P3	+0.1%
Gateway	F1,F3,F4	2,362	2,712	**2,756**	P3-P4	**+16.7%**	P1-P4	+1.6%	2,968	2,882	**3,188**	P3-P4	**+7.4%**	P1-P4	**+10.6%**
Gnrc_networking	F1,F3,F4	421	416	**668**	P3-P4	**+58.7%**	P3-P4	**+60.6%**	1,849	1,779	**2,136**	P3-P4	**+15.5%**	P3-P4	**+20.1%**
GPSTracker	F1,F2,F3,F4	661	977	**1,011**	P2-P4	**+53.0%**	P2-P4	+3.5%	1,227	1440	**1,589**	P2-P4	**+29.5%**	P2-P4	**+10.3%**
Heat_Press	F1,F2,F3,F4	551	555	**570**	P2-P4	**+3.4%**	P2-P4	**+2.7%**	573	580	**601**	P2-P4	+4.9%	P2-P4	+3.6%
L2cap_Processor	F3	**1,001**	1,001	**1,001**	P3	+0.0%	P3	+0.0%	1,002	1,021	**1,170**	P3	**+16.8%**	P3	**+14.6%**
LiteOS_IoT	F1,F3,F4	738	746	**1,333**	P3-P4	**+80.6%**	P1-P4	**+78.6%**	1,375	1,380	**1,377**	P3-P4	**+0.1%**	P1-P4	-0.2%
PLC	F1,F2,F3	638	640	642	P2-P3	+0.3%	P2-P3	+0.3%	640	640	**1,838**	P2-P3	**+187.2%**	P2-P3	**+187.2%**
Filesystem	F1,F3,F4	-	-	-	-	-	-	-	1,374	1,352	**1,414**	P3	+2.9%	P3	+4.6%
Snmp_Server	F3	1,032	1,032	**1,045**	P3	+1.3%	P3	+1.3%	1,066	1,083	**1,297**	P3	**+21.7%**	P3	**+19.8%**
Soldering_Iron	F1,F3,F4	2,177	**2,280**	2,267	P3	+4.1%	P3	-0.6%	2,675	2,799	**3,271**	P3	**+22.3%**	P3	**+16.9%**
Steering_Control	F1,F3,F4	587	594	**606**	P3	+3.1%	P3	+1.9%	652	655	**660**	P3	+1.2%	P3	+0.8%
Zephyr_SocketCan	F1,F3	2,583	**2,662**	2,660	P3	+3.0%	P3	-0.1%	3,334	2,880	**3,341**	P3	+0.2%	P3	**+16.0%**
Client-Gattupdate	F1,F3,F4	-	-	-	-	-	-	-	4,333	3,051	**7,454**	P3-P4	**+72.0%**	P3-P4	**+144.3%**
Server-Gattupdate	F1,F3,F4	-	-	-	-	-	-	-	10,117	10,846	**11,018**	P3-P4	+8.9%	P3	+1.6%
BLE-HCI	F1,F3,F4	2,523	2,576	**2,784**	P3-P4	+10.3%	P3-P4	**+8.1%**	-	-	-	-	-	-	-

-: The original firmware fuzzer fails to reach the stage where firmware receives external input (*e.g.,* getting stuck in initialization stage after 24 hours).

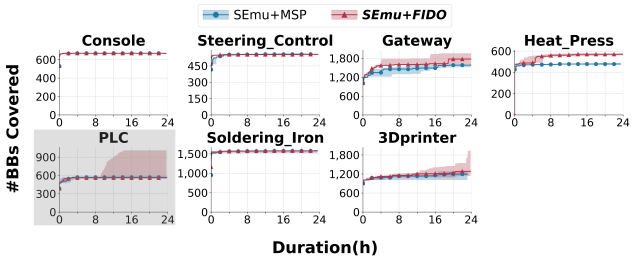


Figure 7: Code coverage comparison between *SEmu*+MSP and *SEmu+FIDO* (only 7 samples are shown, as the other samples require peripheral models that *SEmu* does not extract).

fuzz delivery strategies, each byte is delivered after certain number of basic blocks are executed, allowing enough time for inputs to be read, so the overfeeding problem P1 is less common, except for input routes containing stricter length checks like `BLE_Bootstrap`

As illustrated in Figure 7, *FIDO* achieves greater coverage than *SEmu*, which relies on manually specified delivery points. While *SEmu* can address P2 and P3 if the delivery points match those identified by *FIDO*, as seen with `Console`, it cannot infer quantity requirements. This limitation results in delivering all input at once without length restrictions, leading to the P1 problem across all test samples. Additionally, for firmware with multiple input routes like `Heat_Press`, `Gateway`, and `Steering_Control`, *SEmu* supports input delivery at only a single point, leaving P4 unresolved.

6.2.2. Bug Finding Capability Enhancement. *FIDO* not only improves coverage but also enhances bug-finding capabilities. We extend the fuzzing test period to over 48 hours to assess the improvement. As shown in Table 4, *FIDO* triggered crashes more frequently and faster, with some cases being over 10 times quicker than the original method. For example, `Bootstrap (SPI)` has an out-of-bound-write vulnerability (CVE-2020-10065) in BLE HCI `BT_BUF` buffer. Triggering the corresponding crash requires over 77 bytes in a single command to overflow the `BT_BUF` buffer. If a command is received before HCI initialization or during the last command processing stage, it aborts the current command and responds with an error(`HCI_ERROR_CMD_DISALLOWED`). However, the RR or fuzz delivery strategy used by *MULTIFUZZ* and *Fuzzware* delivers only one byte of input at uncertain time, often causing the HCI controller to abort the retrieved command. This reduces the chance of accumulating a long input that could cause a buffer overflow, making it difficult for the fuzzer to detect this vulnerability. In contrast, *FIDO* delivers all inputs based on firmware requirements (*i.e.,* at checking HCI command availability point), allowing longer command retrieval and increasing the likelihood of triggering the vulnerability (7min to trigger with *FIDO* vs. 7h with RR).

Analysis of newly Discovered Bugs. *FIDO* uniquely detected ten bugs (two for *SEmu*, two for *Fuzzware*, and six for *MULTIFUZZ*) including five 0-day bugs missed by RR and Fuzz delivery methods even in prior work's extensively fuzzed samples.

In the `Gateway`, two new bugs were found in the I2C input route processing functions. The first is a buffer overflow during decoding of longer messages. The second is a NULL pointer dereference, occurring when a command lacks a parameter field, causing the input buffer allocation of parameter field to return NULL and leading to a system crash due to uninitialized NULL pointer dereference. *FIDO* can uniquely identify these bugs by enabling input processing for I2C routes through solving P4, as mentioned later, while RR and Fuzz delivery failed. The unique detection of known bugs in `CCN-Lite-Relay`, `Heat_Press`, and `PLC` also shares the same reasons.

TABLE 4: Crash detection comparison with and without *FIDO* over 48 hours: Crash count represents the total unique crashes, excluding false positives. Time is denoted as (hh:mm:ss). Newly discovered vulnerability is in green bold text. Fac. shows the reduction factor in time when using *FIDO* for bug discovery compared to the original method and have significant changes are marked in bold (based on a Mann-Whitney U test with a 0.05 significance threshold).

Firmware	Bug[Report] Type-Func./CVE-20..	# Crash Count Original	# Crash Count w.*FIDO*	Minium Discovery Time Original	Minium Discovery Time w.*FIDO*	Fac.
***SEmu* (Original = MSP)**						
Heat_Press	OOB-FC3 [43]	Fail	**175**	-	00:09:31	**>302**
PLC	OOB-FC1 [44]	24	**112**	03:23:18	00:44:24	**4.58**
	OOB-FC3 [47]	Fail	**17**	-	01:56:40	**>24**
	OOB-FC15 [45]	135	**175**	00:14:23	00:08:52	**1.62**
	OOB-FC16 [46]	100	**185**	01:53:08	00:33:48	**3.35**
***Fuzzware* (Original = The better results of RR and Fuzz)**						
Heat_Press	OOB-FC3 [43]	1236	**1765**	00:56:12	00:19:52	**2.83**
PLC	OOB-FC1 [44]	258	**399**	00:42:35	00:46:27	0.92
	OOB-FC3 [47]	68	**129**	05:07:08	04:48:25	1.06
	OOB-FC15 [45]	244	**385**	00:10:23	00:09:56	1.05
	OOB-FC16 [46]	88	**1329**	01:28:57	00:21:26	**4.15**
Echo_Server	21-3319 [25]	1590	**1620**	00:27:34	00:25:13	1.09
	21-3320 [26]	1	**18**	02:33:05	03:42:53	0.69
	20-10064 [22]	2	**8**	10:34:24	11:30:48	0.92
Bootstrap(UART)	21-3329 [27]	Fail	**1**	-	35:26:12	>1.41
Bootstrap(SPI)	20-10065 [23]	66	**774**	07:31:43	00:07:02	**64.23**
	20-10066 [24]	15	**225**	05:18:11	03:45:45	**1.41**
BLE-HCI	**NPD-ull_conn.**	Fail	**1**	-	08:16:22	**>6.04**
***MultiFuzz*(Original = The better results of RR and Fuzz)**						
Heat_Press	OOB-FC3 [43]	127	**302**	00:04:02	00:01:12	**3.36**
PLC	OOB-FC1 [44]	91	**374**	00:00:41	00:00:20	**2.05**
	OOB-FC3 [47]	7	**15**	04:44:55	04:31:55	1.05
	OOB-FC15 [45]	6	**9**	00:26:22	00:02:34	**10.27**
	OOB-FC16 [46]	6	**20**	00:28:17	00:26:24	1.07
GPSTracker	NPD-strtok_xx [10]	5	**5**	02:01:33	01:28:02	1.38
	NPD-strstr_xx [11]	Fail	**3**	-	05:01:49	**>9.6**
	NPD-strstr_xx [12]	2	**3**	09:51:50	05:45:03	1.72
Gateway	ING-Sysex. [6]	6	**8**	00:03:55	00:05:46	0.68
	OOB-setPin. [40]	**30**	20	00:02:39	00:01:51	1.43
	UPD-TxCplt [41]	**10**	8	00:00:01	00:00:01	1
	NPD-pwm_start [42]	97	**764**	00:17:20	00:13:07	1.32
	OOB-decode.	Fail	**1**	-	08:00:43	**>6**
	NPD-processSysex.	Fail	**5**	-	06:30:13	**>7.38**
Echo_Server	20-10064 [22]	90	**95**	03:23:34	01:38:02	**2.08**
Bootstrap(UART)	**NPD-net_buf_simple.**	Fail	**1**	-	05:50:42	**>8.23**
Bootstrap(SPI)	20-10065 [23]	10	**15**	00:11:28	00:02:35	**4.44**
L2cap_Processor	20-12140 [38]	Fail	**5**	-	00:53:16	**>57**
Snmp_Server	20-12141 [39]	**3**	2	08:57:22	06:21:03	1.41
CCN-Lite-Relay	RC-ble_isr [13]	20	**25**	00:01:12	00:01:46	0.68
	NPD-ccnl [8]	Fail	**2**	-	18:35:26	**>2.69**
	NPD-evtimer [9]	4	**5**	09:50:23	06:11:44	1.59
	UAF-ccnl_xx [15]	20	**28**	03:49:21	01:48:32	**2.11**
SocketCan	NPD-net_pkt [14]	6	**8**	16:10:02	12:37:22	1.28
	NPD-pwm_shell	3	**9**	17:42:07	05:08:16	**3.45**
Gattupdate*	**24-22095**	40	**43**	07:27:16	00:15:32	**28.79**

OOB: Out of Bound Access; ING: Integer Overflow N/UPD: NULL/Uninitialized Pointer Deference; UAF:Use After Free; RC:Race Condition
*:`Client-Gattupdate` and `Server-Gattupdate` samples yield very similar results outcomes, so we present the average result.

The `Client/Server-Gattupdate` samples vulnerability is in BLE Cordio implementation of Arm Mbed OS 6.17.0, occur when an invalid packet type is received, leading to buffer overflow due to data accumulation in a constrained while loop. The *FIDO*-enable fuzzer effectively delivers inputs in a stable pattern, allowing long inputs to fill in the buffer and trigger overflow. In contrast, the RR and Fuzz delivery method feeds inputs separately at incorrect timings, causing frequent buffer resets due to error handling, thus reducing the likelihood of triggering the overflow.

The `BLE-HCI` vulnerability in the BLE subsystem implementation of Zephyr 4.1.0 arises from an uninitialized connection address variable in the memory pool during BLE initialization, leading to a device crash when dereferenced later in the command response process (`tx_demux`). To trigger this crash, BLE must complete initialization and retrieve the command first. *FIDO* uniquely detected this bug by enabling more times and longer input retrieval, similar to the CVE-2020-10065.

The `BLE-Bootstrap(UART)` vulnerability occurs when the allocated space for input data exceeds the remaining space, causing the allocation to fail and return a NULL pointer buffer, leading to a NULL pointer dereference. Under RR and Fuzz, the delivery interval is long, making it difficult to input to accumulated in heap. However, *FIDO* can deliver longer input at once at delivery point, allowing this bug to be detected.

TABLE 5: Issues with *AidFuzzer* under frequency 1 (F=1) and frequency 10 (F=10).(-: The execution is stuck.)

Firmware	Unsatisfied Input Route Perip.:#:[Lower:Upper],...	F = 1 P1	F = 1 P2	F =10 P3	F =10 P4
Console	UART:1:[1:64]	No	No	Yes	No
Steering_Control	UART:2:[1:128]	Yes	No	Yes	No
Gateway	UART:1:[1:64],I2C:2:[1:128]	No	No	Yes	Yes
Heat_Press	UART:6:[8:64]	-	Yes	Yes	No
PLC	UART:1:[8:64]	-	Yes	Yes	No
Soldering_Iron	I2C:1:[1:128]	No	No	Yes	No
GPS_Tracker	UART:1:[1:256],UART:4:[1:256]	No	No	Yes	Yes
LiteOS_IoT	UART:1:[1:100],UART:5:[1:100]	No	No	Yes	Yes
3Dprinter	UART:1:[1:64],USB:1	No	No	Yes	No
SocketCan	CAN:1:[1:64]	No	No	Yes	No
μTasker_USB	USB:1:[1:512]	No	No	Yes	No
Bootstrap(UART)	UART:1:[1:5]	Yes	No	Yes	No
Bootstrap(SPI)	SPI:1:[1:5]	Yes	No	Yes	No
Echo_Server	SPI:1:[1:132]	No	No	Yes	No
L2cap_Processor	RADIO:1:[1:128]	No	No	Yes	No
Snmp_Server	RADIO:1:[1:128]	No	No	Yes	No
CCN-Lite-Relay	UART:1:[1:128],Radio:1:[1:168]	No	No	Yes	Yes
Gnrc_networking	UART:2:[1:128]	No	No	Yes	No
Client-Gattupdate	SPI:2:[1:256]	No	No	Yes	No
Server-Gattupdate	SPI:2:[1:256]	No	No	Yes	No
BLE-HCI	UART:1:[1:7]	Yes	No	Yes	No

6.3. Comparison to Interrupt-driven Firmware Fuzzers (RQ3)

A recent work, *AidFuzzer* [54], introduces an interrupt-driven test-case delivery mechanism, aiming to address a problem similar to that of *FIDO*. The main observation of *AidFuzzer* is that firmware often enters waiting states, such as when encountering sleep instructions like `WFI`, entering an infinite loop, or reading a global variable modifiable in an ISR. *AidFuzzer* delivers input when a waiting state is detected. While *FIDO* is designed as a drop-in replacement for the input delivery mechanism of existing fuzzers, we found it challenging to integrate *FIDO* with *AidFuzzer*'s underlying emulator due to the tight coupling of *AidFuzzer*'s interrupt mechanism with its emulator.

To compare *FIDO* with *AidFuzzer*, we conducted two experiments. First, we tested 27 samples using *AidFuzzer* and compared the results with *FIDO* using *MULTIFUZZ*.

After 24 hours, *AidFuzzer* produced output for 16 samples, while execution failed for others due to unsupported memory map alignment or initial seed crashes. For the 16 successful samples, *FIDO* with *MULTIFUZZ* consistently achieved significantly higher code coverage, ranging from 5% to over 1,500%, as detailed in Appendix F.

In our second experiment, we re-implemented one of *AidFuzzer*'s waiting-state detections (reading a global variable modifiable in an ISR) and integrated it into

TABLE 6: Ablation study for the effect of S1-S3, using *Fuzzware* in RR mode as the baseline. ↑ indicates changes in median coverage and average crash count compared to the previous configuration. Changes below 0.1% are not displayed, and significant changes are marked in bold (based on a Mann-Whitney U test with a 0.05 significance threshold).

Firmware	RR		S1					+S2					+S3				
	Cov.	Crash	Coverage			Crash		Coverage			Crash		Coverage			Crash	
	Med.	Count	Med.	↑	*p-value*	Count	↑	Med.	↑	*p-value*	Count	↑	Med.	↑	*p-value*	Count	↑
Gateway	2362	0	2,512	+6.4%	0.056	0		2,558	+1.8%	0.548	0		2,756	+7.7%	0.095	0	
Heat_Press	551	247.2	563	+2.2%	0.010	235.6	-4.7%	565	+0.4%	0.666	259	+9.9%	570	+0.9%	0.916	353	+36.3%
CCN-Lite-Relay	491	0	1,054	+114.7%	0.011	0		1,054		0.796	0		1,054		1.000	0	
Gnrc_Networking	421	0	666	+58.2%	0.014	0		666		0.821	0		668	+0.3%	0.564	0	
3Dprinter	786	0	902	+14.8%	0.056	0		901		0.916	0		931	+3.3%	0.140	0	
GPSTracker	661	0	948	+43.4%	0.032	0		952	+0.4%	1.000	0		1,011	+6.2%	0.012	0	
LiteOS_IoT	738	0	739	+0.1%	0.083	0		1,079	+46.0%	0.408	0		1,333	+23.5%	0.292	0	

MULTIFUZZ. Additionally, *AidFuzzer* employs an opportunistic strategy with a configurable triggering frequency, randomly skipping some waiting-state encounters. Our configuration covers both the minimum and maximum frequencies (1/10). The results are summarized in Table 5.

Problems under Minimum Frequency. When the ISR trigger configuration time is set to one, *AidFuzzer* delivers input whenever a waiting state is encountered, without considering the buffer's maximum capacity, which causes P1. For example, `Steering_Control` reads input until it encounters a comma, line break, or an empty buffer. Before each read, it checks global variables. Consequently, *AidFuzzer* continues to deliver input before buffer reading, regardless of whether the input length exceeds the buffer capacity. Additionally, as mentioned in F2, some firmware has buffer cleaning behavior ($\widetilde{R_B}$), and input should not be delivered at $\widetilde{R_B}$, as it would be wasted. However, if the frequency is one, *AidFuzzer* can avoid this issue. For instance, in Listing 1, *AidFuzzer* continuously feeds new data into the buffer at the `Head` pointer reading, causing the firmware to become stuck in the data retrieval loop and waste input (P2) (Line 59).

Problems under Maximum Frequency. For *AidFuzzer*, if the configuration frequency exceeds 1, the firmware enters the waiting state at least twice, but only one time interrupt is triggered (*i.e.,* typically, one byte is delivered per interrupt). This results in failed availability checks in P3 and P4. Taking `Heat_Press` as an example in Listing 1, since the `Head` pointer is modified in the ISR, *AidFuzzer* interprets `Head` pointer readings in functions like `available` as waiting states. If the waiting state is set to ten, only one byte can be retrieved from the ISR buffer after ten availability checks at Line 59, preventing the length check from passing. Additionally, not all availability checks for each input route are unconditional repeated; some routes are checked only under specific conditions, such as in `Gateway`, `GPS_Tracker`, and `CCN-Lite-Relay`. A high-frequency configuration (with less interrupt triggering) can lead to P4 problems.

In summary, we found that the coarse-grained input delivery reasoning in *AidFuzzer* mitigates the issue with random input delivery to some extent, but it cannot effectively address all the identified problems in this paper.

6.4. Ablation Study (RQ4)

To quantify the marginal contribution of each component, including delivery point timing inference (S1), length-range inference (S2), and multi-route-aware delivery coordination (S3), we conduct ablation studies using fuzzing metrics (*i.e.,* basic block coverage and crash counts). These studies are performed on representative samples involving multiple input routes (`3DPrinter`, `GPSTrackers`, `CCN-Lite-Relay`, `Gnrc_Networking`, `Gateway`, `Heat_Press`, and `LiteOS_IoT`; see details in Table 10 in the Appendix) to demonstrate the impact of each component. To isolate the effects of each component, we start with a baseline configuration using Fuzzware in RR delivery mode. We then construct three configurations for the ablation study, each incrementally adding one component of *FIDO*. The first configuration includes only S1, the second adds S2 to S1, and the third includes all three components (S1, S2, and S3). The results are shown in Table 6.

+S1: We utilize delivery point information extracted by S1 for fuzzing, but deliver only one byte at each point via interrupt. Enabling S1 improves coverage by 34.3% on average by avoiding P4. In contrast, interrupt delivery via round-robin in *Fuzzware* is less effective because most interrupt triggering does not deliver any byte (*e.g.,* `Gateway`, `CCN-Lite-Relay`, and `GPSTracker`, detailed in Appendix E). For samples with length checks (*e.g.,* `LiteOS_IoT` with a 20B limit and `Heat_Press` with 8B), S1 alone requires more time to reach the limit, resulting in less coverage improvement and slightly fewer crashes than the round-robin mode for `Heat_Press`.

+S2: Building on S1, S2 ensures that the delivery length meets the required ranges, effectively handling samples such as `LiteOS_IoT` and `Heat_Press` with length checks. This increases coverage for `LiteOS_IoT` by 46% and crash counts for `Heat_Press` by 10% compared to S1 alone. For other samples, length checks are not applicable; therefore, S2 does not provide improvement. Without a delivery schedule (S3), S2 alone may introduce negative effects because the distribution of input routes per fuzzing round becomes unpredictable, leading to uneven input distribution (*e.g.,* `LiteOS_IoT` and `Heat_Press`).

+S3: Enabling S3 offers two benefits: (1) maximizing coverage by testing all input routes (achieving a 6% average increase on top of S1+S2), and (2) mitigating the negative effects of S2 by balancing input distribution. For example,

a vulnerability in `Heat_Press` was found in one of six routes. The input scheduling algorithm Algorithm 1 reserves a minimum input length for each route, ensuring stable delivery across all routes and increasing crash numbers by 36% compared to S1+S2.

7. Limitations and Discussion

Non-IO Handling. Our design focuses on how firmware handles inputs from real peripherals, meaning that fuzzing inputs should only be consumed by external data register reads. Internal peripheral registers, such as status registers, should be managed by real hardware or emulation. *FIDO* relies on existing firmware fuzzers in handling internal peripheral registers, and thus inherits their limitation to use fuzzing inputs for both data register reads and some internal hardware register reads. This can lead to minor inaccuracies in sub-input length calculation in Step 3, causing P1–P4 problems to occur.

Orthogonality with Existing Front-end Optimizations. In the fuzzer front-end, the proposed input delivery is not entirely orthogonal from existing front-end optimizations such as input-to-state (I2S) and length extension. They all influence whether mutated inputs reach and exercise the processing logic, and their interaction can be complex. For instance, an I2S mutation might adjust byte positions for long-string comparisons; if delivery timing and length change, the effective byte positions may shift, reducing effectiveness. It is our future work to more thoroughly study the interactions when integrating delivery information into existing fuzzing optimizations.

DMA Support. High-throughput peripherals like USB and Ethernet commonly use DMA to allow data transfers between RAM and peripherals without processor involvement. *FIDO* as a plugin for *MULTIFUZZ*, *Fuzzware* and *SEmu*, does not directly support emulating DMA peripherals. To support DMA, we need to identify the delivery point for DMA transactions. Inspired by GDMA [49], we found that *FIDO*'s analysis can be ported to achieve DMA delivery point identification, with some chip-specific knowledge. For example, the six common DMA configurations summarized in GDMA can also be mapped to two notification modes similar to polling and interrupt modes.

In the polling mode, after a DMA transaction, specific fields in MMIO registers (MMIO-based DMA Configuration) or DMA descriptors (RAM-based DMA Configuration) are updated. Firmware checks this field to determine input availability, similar to polling mode checks. In the interrupt mode, DMA can be set to trigger an interrupt upon transaction completion, updating global variables. Firmware then checks these variables to confirm transaction completion, similar to interrupt mode. Therefore, we can reuse *FIDO* to identify global variable checking points as the delivery points for DMA data. However, determining delivery length, usually indicated by specific fields in MMIO registers or descriptors, requires precise DMA modeling like GDMA, which our length inference method cannot achieve.

Generalizability. While the idea of delivery information extraction is general to firmware, its implementation is specific to the underlying fuzzer's front-end or back-end.

8. Related Work

Firmware Dynamic Analysis. Dynamic analysis techniques, particularly fuzzing, are effective in identifying bugs in various software [62]. However, applying these techniques to MCU firmware is challenging due to the reliance on resource-constrained hardware and the lack of source code. Some researchers have tried integrating these techniques with original hardware [31], [32], [33], [34], [53], [57]. Testing with physical hardware is challenging for scaling and is often ineffective. Consequently, recent efforts focus on developing effective re-hosting environments for firmware fuzzing [16], [17], [21], [28], [48], [52], [59]. For instance, *Fuzzware* [48] and *μEmu* [59] used symbolic execution to derive peripheral MMIO models from firmware behaviors for emulating peripheral reads. *SEmu* [60] and *Perry* [16] created peripheral models by extracting hardware logic from public manuals or peripheral driver code, to increase the emulation fidelity.

On the other hand, recent researchers have begun adapting fuzzing techniques to incorporate these features. *EmberIO* [18] remaps edge-coverage feedback to eliminate invalid new edges caused by random interrupts, while *SplITS* [19] addresses multi-byte magic strings in firmware to uncover new code coverage faster. *Hoedur* [50] and *MULTIFUZZ* [7] adapt general fuzzing techniques, including input generation, mutation, and feedback, to account for the multi-stream nature of firmware inputs from various peripherals. In comparison, we identify asynchronous input handling as a new feature impacting firmware fuzzing and propose *FIDO* to improve input delivery. In addition, since *FIDO* focuses solely on enhancing input delivery, it can be used alongside other state-of-the-art firmware fuzzing optimizations and emulations. *AIM* [20] predicts interrupt-firing timing using persistent symbolic execution, but it is incompatible with fuzzing and incurs significant overhead from symbolic execution. *AidFuzzer* [54] suggests a waiting state-based interrupt firing solution. However, delivering at every waiting state may stuff the firmware, whereas delivering too infrequently may starve it. Moreover, *AidFuzzer* does not attempt to determine the proper amount of data to provide, leaving the delivery quantity undefined.

Firmware Static Analysis. Static analysis typically used in firmware security analysis to identify specific vulnerabilities. For example, Firmalice [51] and PASAN [29] target authentication bypass and race condition and peripheral access. SaTC [4] employs static data-flow analysis to detect taint-style vulnerabilities by identifying user input through shared keywords. Additionally, static analysis aids firmware fuzzing; for instance, SFuzz [3] uses forward slicing to prune paths that are independent of external inputs, ad-

dressing firmware emulation challenges. Our tool, *FIDO*, aligns with this approach by utilizing static data and control flow analysis to identify optimal input delivery timing and quantity for fuzzing.

9. Conclusion

In this work, we found that the delivery method also impacts effectiveness and efficiency due to asynchronous interactions between peripheral input arrival and firmware input handling, a factor often overlooked. To identify the optimal delivery time and quantity, We developed, *FIDO*, which automatically extracts delivery information—such as delivering points and expected input volume range—by identifying and analysis the semantic of key input handling operations in firmware programming model (check-retrieval-processing, CRP) through static and dynamic analysis. Integrated with SOTA firmware fuzzer, we show that *FIDO* significantly improves fuzzing by increasing coverage and capability of bug detection compared to ad-hoc delivery method such as periodic, fuzz and MSP pattern. Our findings highlight the importance of firmware-aware input delivery mechanisms in firmware fuzzing and open new area for firmware fuzzing improvement.

Ethics Considerations

Our tool identifies vulnerabilities in MCU-based device firmware using fuzzing. We conduct experiments on firmware in emulators within an isolated internal server. All security bugs found in this work have been reported to vendors/developers as detail in Appendix D. The firmware images used in our study were sourced from public resources.

Acknowledgment

We sincerely appreciate our shepherd and all the anonymous reviewers for their insightful and valuable feedback. This work was supported by National Natural Science Foundation of China (NSFC) grant (62202188).

References

[1] N. S. Agency, "Ghidra," https://ghidra-sre.org/, 2023, last accessed: 2024-11-1.

[2] ——, "Ghidra-Server.org provides a collaboration server on the internet for the software reverse engineering," https://www.ghidra-server.org/, April 2025.

[3] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang *et al.*, "Sfuzz: Slice-based fuzzing for real-time operating systems," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 485–498.

[4] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 303–319.

[5] M. Chesser, S. Nepal, and D. C. Ranasinghe, "Icicle: A re-designed emulator for grey-box firmware fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 76–88.

[6] ——, "Incorrect handling of zero length sysex messages," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#incorrect-handling-of-zero-length-sysex-messages, 2024, last accessed: 2024-03-01.

[7] ——, "Multifuzz: A multi-stream fuzzer for testing monolithic firmware," in *33rd USENIX Security Symposium (USENIX Security)*, 2024.

[8] ——, "Null Pointer Dereference in ccnl_cs," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#issue-with--encoded-characters-in-ccnl_cs, 2024, last accessed: 2024-03-01.

[9] ——, "Reinitialization of shared global timer," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#reinitialization-of-shared-global-timer, 2024, last accessed: 2024-03-01.

[10] ——, "Stdio initialization race," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#return-value-of-strtok-not-checked-for-null-in-gsm_get_imei, 2024, last accessed: 2024-03-01.

[11] ——, "Stdio initialization race," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#return-value-of-strstr-not-checked-for-null-in-sms_check, 2024, last accessed: 2024-03-01.

[12] ——, "Stdio initialization race," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#return-value-of-strstr-not-checked-for-null-in-gsm_get_time, 2024, last accessed: 2024-03-01.

[13] ——, "Stdio initialization race," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#stdio-initialization-race, 2024, last accessed: 2024-03-01.

[14] ——, "Stdio initialization race," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#net-pkt-command-dereferences-a-user-provided-pointer, 2024, last accessed: 2024-03-01.

[15] ——, "Use After Free in evtimer struct," https://github.com/MultiFuzz/MultiFuzz-benchmarks/blob/main/crash-analysis.md#missing-removal-from-evtimer-struct, 2024, last accessed: 2024-03-01.

[16] L. Chongqing, L. Zhen, Z. Yue, Y. Yan, L. Junzhou, and F. Xinwen, "A friend's eye is a good mirror: Synthesizing mcu peripheral models from peripheral driver," in *USENIX Security*, 2024.

[17] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "{HALucinator}: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1201–1218.

[18] G. Farrelly, M. Chesser, and D. C. Ranasinghe, "Ember-io: Effective firmware fuzzing with model-free memory mapped io," in *Proceedings of the 2023 ACM Asia conference on computer and communications security*, 2023.

[19] G. Farrelly, P. Quirk, S. S. Kanhere, S. Camtepe, and D. C. Ranasinghe, "Splits: Split input-to-state mapping for effective firmware fuzzing," in *European Symposium on Research in Computer Security*. Springer, 2023, pp. 290–310.

[20] B. Feng, M. Luo, C. Liu, L. Lu, and E. Kirda, "Aim: Automatic interrupt modeling for dynamic firmware analysis," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[21] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of Usenix Security Symposium*, 2020.

[22] L. Foundation, "CVE-2020-10064 Description," https://docs.zephyrproject.org/latest/security/vulnerabilities.html#cve-2020-10064, 2020, last accessed: 2024-03-01.

[23] ——, "CVE-2020-10065 Description," https://docs.zephyrproject.org/latest/security/vulnerabilities.html#cve-2020-10065, 2020, last accessed: 2024-03-01.

[24] ——, "CVE-2020-10066 Description," https://docs.zephyrproject.org/latest/security/vulnerabilities.html#cve-2020-10066, 2020, last accessed: 2024-03-01.

[25] ——, "CVE-2021-3319 Description," https://github.com/zephyrproject-rtos/zephyr/security/advisories/GHSA-94jg-2p6q-5364, 2021, last accessed: 2024-03-01.

[26] ——, "CVE-2021-3320 Description," https://docs.zephyrproject.org/latest/security/vulnerabilities.html#cve-2021-3320, 2021, last accessed: 2024-03-01.

[27] ——, "CVE-2021-3329 Description," https://github.com/zephyrproject-rtos/zephyr/issues/39549, 2021, last accessed: 2024-03-01.

[28] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 135–150.

[29] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu, and D. J. Tian, "{PASAN}: Detecting peripheral access concurrency bugs within {Bare-Metal} embedded applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 249–266.

[30] P. Koopman, "Why short interrupt service routines matter," https://betterembsw.blogspot.com/2013/04/why-short-interrupt-service-routines.html, 2025, last accessed: 2025-06-01.

[31] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μafl: non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1–12.

[32] C. Liu, A. Mera, E. Kirda, M. Xu, and L. Lu, "{CO3}: Concolic co-execution for firmware," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5591–5608.

[33] A. Mera, C. Liu, R. Sun, E. Kirda, and L. Lu, "{SHiFT}: Semi-hosted fuzz testing for embedded applications," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5323–5340.

[34] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.

[35] ——, "Avatar2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, vol. 18, 2018, pp. 1–11.

[36] M. OS., "Official mbed ble examples," https://github.com/ARMmbed/mbed-os-example-ble.

[37] T. Ryan, Asmita, J. Doreen, S. Soheil, M. Prasant, and H. Houman, "Ffxe: Dynamic control flow graph recovery for embedded firmware binaries," in *33rd USENIX Security Symposium (USENIX Security)*, 2024.

[38] T. Scharnowski, "CVE-2020-12140 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/blob/main/03-fuzzing-new-targets/bug-details/CVE-2020-12140-Contiki-NG-l2cap-frame-size.md, 2022, last accessed: 2024-03-01.

[39] ——, "CVE-2020-12141 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/blob/main/03-fuzzing-new-targets/bug-details/CVE-2020-12141-Contiki-NG-SNMP-string-decode.md, 2022, last accessed: 2024-03-01.

[40] ——, "P2IM Gateway OOB write in HAL Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/12, 2022, last accessed: 2024-03-01.

[41] ——, "P2IM Gateway OOB write in HAL Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/21, 2022, last accessed: 2024-03-01.

[42] ——, "P2IM Gateway OOB write in HAL Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/23, 2022, last accessed: 2024-03-01.

[43] ——, "P2IM Heat_Press Bug in get_FC3 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/13, 2022, last accessed: 2024-03-01.

[44] ——, "P2IM PLC Bug in Process_FC1 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/14, 2022, last accessed: 2024-03-01.

[45] ——, "P2IM PLC Bug in Process_FC15 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/16, 2022, last accessed: 2024-03-01.

[46] ——, "P2IM PLC Bug in Process_FC16 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/17, 2022, last accessed: 2024-03-01.

[47] ——, "P2IM PLC Bug in Process_FC3 Description," https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main/04-crash-analysis/15, 2022, last accessed: 2024-03-01.

[48] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[49] T. Scharnowski, S. Hoffmann, M. Bley, S. Wörner, D. Klischies, F. Buchmann, N. O. Tippenhauer, T. Holz, and M. Muench, "Gdma: Fully automated dma rehosting via iterative type overlays," in *34rd USENIX Security Symposium (USENIX Security 25)*, 2025.

[50] T. Scharnowski, S. Woerner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "Hoedur: Embedded firmware fuzzing using multi-stream inputs," in *32nd USENIX Security Symposium (USENIX Security 23)*. Boston, MA: USENIX Association, Aug. 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/scharnowski

[51] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[52] C. Spensky, A. Machiry, N. Redini, C. Unger, G. Foster, E. Blasband, H. Okhravi, C. Kruegel, and G. Vigna, "Conware: Automated modeling of hardware peripherals," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 95–109.

[53] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *27th USENIX Security Symposium*, 2018, pp. 291–307.

[54] J. Wang, Q. Wang, T. Scharnowski, L. Shi, S. Woerner, and T. Holz, "Aidfuzzer: Adaptive interrupt-driven firmware fuzzing via run-time state recognition," in *34rd USENIX Security Symposium (USENIX Security 25)*, 2025.

[55] WikeBooks, "Embedded systems/io programming," https://en.wikibooks.org/wiki/Embedded_Systems/IO_Programming, 2025, last accessed: 2025-07-01.

[56] J. Y. Won, H. Wen, and Z. Lin, "What you see is not what you get: Revealing hidden memory mapping for peripheral modeling," in *25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2022, Limassol, Cyprus, October 26-28, 2022*. ACM, 2022, pp. 200–213. [Online]. Available: https://doi.org/10.1145/3545948.3545957

[57] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, vol. 14, 2014, pp. 1–16.

[58] Zephyr., "Ble hci uart example," https://docs.zephyrproject.org/latest/samples/bluetooth/hci_uart/README.html#bluetooth_hci_uart.

[59] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[60] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, "What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3269–3283.

[61] K. Zhu, Y. Lu, H. Huang, L. Yu, and J. Zhao, "Constructing more complete control flow graphs utilizing directed gray-box fuzzing," *Applied Sciences*, vol. 11, no. 3, p. 1351, 2021.

[62] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

Appendix A. Feature of Unit-test Samples

TABLE 7: Feature of unit-test samples (-:No official demo)

RTOS Driver Demo					
	RIoT	**Nuttx**	**Zephyr**	**MbedOS**	**FreeRTOS**
GPIO	F3	F2, F3	F3	F3	F3
UART	F1, F2, F3	F1, F3	F1, F3	F1, F3	F1, F3
I2C	F1, F3	F1, F3	F1, F3	F1, F3	F1, F2, F3
SPI	-	-	F1, F3	F1, F3	F1, F3
ADC	-	F2, F3	F1, F3	F1, F3	F1, F3
ETH	F2, F3	F2, F3	F3	F3	F3

Bare-Metal SDK Demo					
	STM32F103/F429/L152 HAL		**NXP K64/K66**	**Microchip SAM3**	
	ARDUINO	**STM32Cude**	**HAL SDK**	**ARDUINO**	**HAL SDK**
GPIO	F2, F3	F3	F3	F2, F3	F3
UART	F1, F2, F3	F1, F3	F1, F3	F1, F3	F1, F3
I2C	F1, F2, F3	F1, F3	F1, F3	F1, F2, F3	F1, F2 ,F3
SPI	F1, F3	F1, F3	F1, F3	F1, F3	F1, F2, F3
ADC	F2, F3	F1, F3	F3	F1, F3	F1, F3
ETH	F2, F3	F2, F3	F3	F2, F3	F3

Appendix B. Detail Configuration of Demo in Section 3.2

Configuration-1 used a round-robin delivery method, delivering input every 1,000 basic blocks executed (the default for *Fuzzware*). Configuration-2 employed a fuzz delivery method, with intervals ranging from 1 to 16,000 basic blocks executed, increasing in steps of 250 times powers of 2 from zero to six. One interval is chosen based on the fuzzing input data modulo eight.

For these three groups, the delivery length varied: 1B (the one-time DR read length for the `UART_IRQhandler` function, the default for *Fuzzware*), a random size between 1 and 1,000B (to mimic unconstrained fuzzer-generated input lengths), and a manually restricted length between 8 and 127 (to satisfy minimum length checks).

Configuration-3 uses fixed input delivery points at the start of the main `Loop` function (Line 27 in Listing 1) with a random size between 1 and 1,000B (to mimic unconstrained fuzzer-generated input lengths).

Configuration-4 simulates the ideal situation that delivers inputs at the start of the `Poll` function with sizes randomly selected within the specified range (8 to 127).

Appendix C. Delivery Information of Unit-test Samples

TABLE 8: Delivery information of unit-test samples

RTOS Driver Demos (CRP Route(Perip.:#:[Lower:Upper]))				
	F103/RIOT	**F103/NUTTX**	**K64/RIOT**	**SAM3/RIOT**
GPIO	GPIO:1	GPIO:1	GPIO:1	GPIO:1
UART	UART:1:[1:64]	UART:1:[1:12]	UART:1:[1:64]	UART:1:[1:64]
I2C	I2C:1:[1:4]	I2C:1:[1:32]	-	-
ADC	-	ADC:1:[1:4]	ADC:1:[1:4]	-
Bare-Metal Demos (CRP Route(Perip.:#:[Lower:Upper]))				
	K64/HAL	**K66/HAL**	**F103/Arduino**	**SAM3/Arduino**
GPIO	GPIO:1	GPIO:1	GPIO:1	GPIO:1
UART	UART:1:[1:64]	UART:1:[1:64]	UART:1:[1:128]	UART:1:[1:128]
I2C	I2C:1:[1:100]	I2C:1:[1:100]	I2C:1:[1:100]	I2C:1:[1:100]
ADC	ADC:1:[1:4]	ADC:1:[1:4]	ADC:1:[1:4]	ADC:1:[1:4]

Appendix D. Responsible Disclosure

As of March 31, 2026, the bugs found in `Client/Sever-Gattupdate` and `BLE-HCL` have been fixed. The bug in `Bootstrap(UART)` was acknowledged by the vendor, and a fix is under discussion. Two bugs in `Gateway`, which are only locally exploitable, have not received any response from the vendor.

TABLE 9: Detail of newly discovered Bugs by *FIDO*

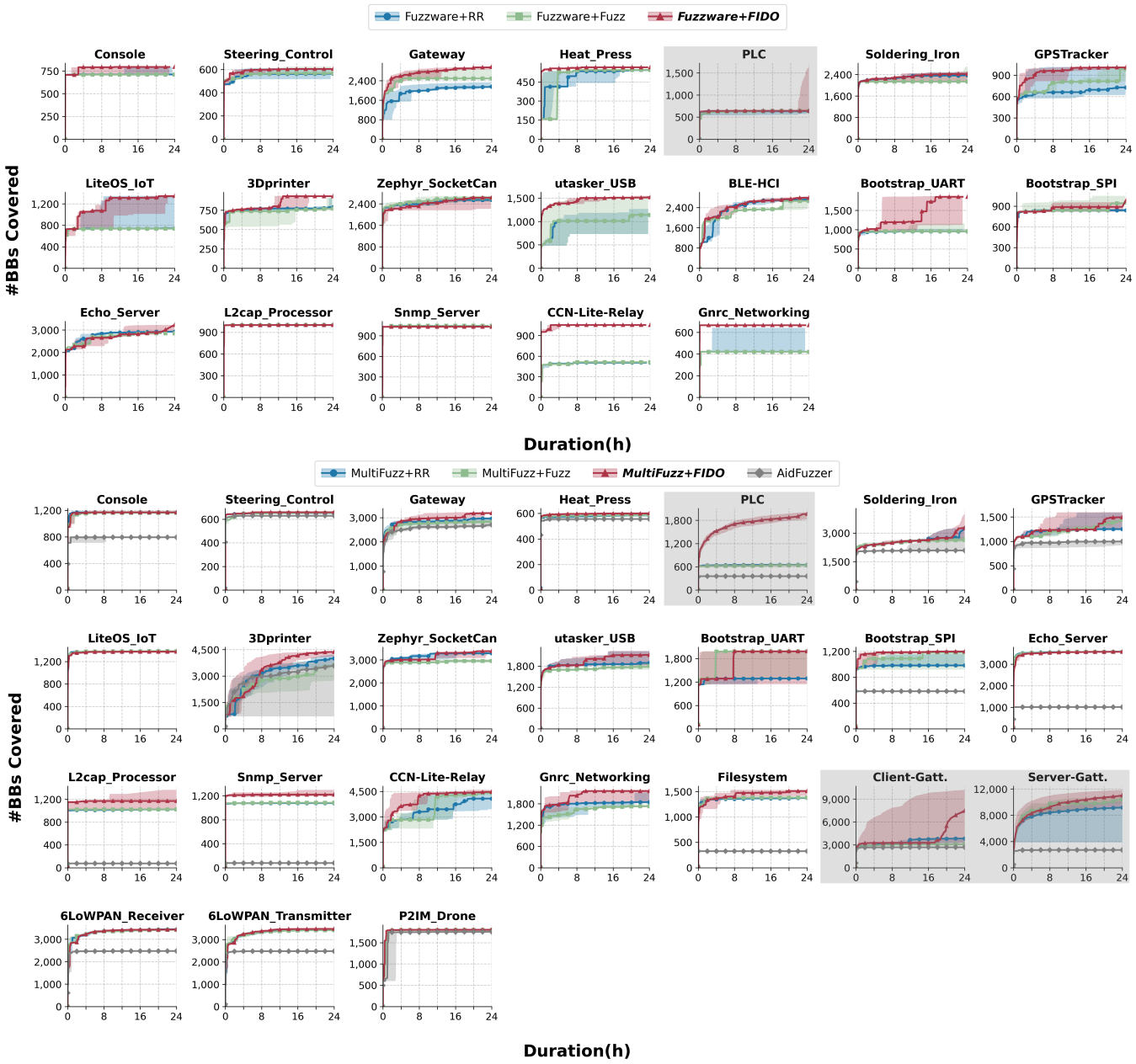
Firmware	Bug Type	Vulnerable Func.	Status
BLE-HCL	NULL Pointer Deference	ull_conn_tx_lll_enqueue	Fixed
Gateway	Out-of-bound-write	decodeByteStream	Reported
Gateway	NULL Pointer Deference	processSysexMessage	Reported
Bootstrap(UART)	NULL Pointer Deference	net_buf_simple_tailroom	Acknowledged
Client/Sever-Gattupdate	Out-of-bound-write	hciTrSerialRxIncoming	Fixed

Appendix E. Details of Real-world Firmware Samples

TABLE 10: Details and input routes of 25 real-world firmware samples (Underline input routes in polling mode, others in interrupt mode. Bold input routes that always occur in main loop; other routes occur only in specific conditions.)

Firmware	MCU	OS/Sys lib.	Total	CRP Route(Peripheral.:#:[Lower:Upper],...))
Console	NXP K64F	NXP HAL	2,251	**UART:1:[1:64]**
Steering_Control	SAM3X	Arduino	1,835	**UART:2:[1:128]**
Gateway	STM32F103	Arduino	4,921	**UART:1:[1:64]**,I2C:2:[1:128],GPIO:4,ADC:1
Heat_Press	SAM3X	Arduino	1,837	**UART:6:[8:64]**
PLC	STM32F429	Arduino	2,304	**UART:1:[8:64]**
Soldering_Iron	STM32F103	FreeRTOS	3,657	**I2C:1:[1:128]**,**GPIO:1**,**ADC:1**
GPS_Tracker	SAM3X	Arduino	4,194	**UART:1:[1:256]**,UART:4:[1:256]
LiteOS_IoT	STM32L431	LiteOS	2,423	**UART:1:[20:100]**,UART:5:[20:100]
3Dprinter	STM32F103	STM32 HAL	8,045	**UART:1:[1:64]**,**USB:1**,GPIO:2
SocketCan	STM32L432	Zephyr	5,943	**CAN:1:[1:64]**
μTasker_USB	STM32F429	μTasker	3,491	UART:1:[1:516],USB:1:[1:512],**GPIO:1**
Bootstrap(UART)	nRF52840	Zephyr	4,972	**UART:1:[1:5]**,GPIO:1
Bootstrap(SPI)	nRF52840	Zephyr	4,949	**SPI:1:[1:5]**,GPIO:1
Echo_Server	SAM4E	Zephyr	7,007	**SPI:1:[1:132]**
L2cap_Processor	TICC2538	Contiki-NG	4,002	**RADIO:1:[1:128]**
Snmp_Server	TICC2538	Contiki-NG	3,080	**RADIO:1:[1:128]**
CCN-Lite-Relay	nRF52832	Nordic HAL	12,675	**UART:1:[1:128]**, Radio:1:[1:168]
Gnrc_networking	STM32F303	STM32 HAL	6,448	**UART:2:[1:128]**
Filesystem	STM32F303	STM32 HAL	2,429	**UART:1:[1:128]**,UART:4:[1:128]
6Lowpan_Receiver	SAM R21	Contiki	6,988	UART:1, **Radio:1**, I2C:1
6Lowpan_Transmitter	SAM R21	Contiki	6,988	UART:1, **Radio:1**, I2C:1
P2IM_Drone	STM32F103	Bare-Metal	2,754	UART:1:[1:2], **I2C:4**
Client-Gattupdate	nRF52840	MBedOS	13,888	**SPI:2:[1:256]**,**GPIO:1**
Server-Gattupdate	nRF52840	MBedOS	13,826	**SPI:2:[1:256]**,**GPIO:1**
BLE-HCI	nRF52840	Zephyr	7,470	**UART:1:[1:7]**

Appendix F.
Detail of Fuzzing Coverage Comparison Results



Note: The fuzzing process with *AidFuzzer* fails to start for `LiteOS_IoT`, `Gnrc_networking`, `µtasker_USB`, `Zephyr_SocketCan`, `Bootstrap(UART)`, and `BLE-HCI`.

Figure 8: Code Coverage Comparison with and without *FIDO* on *Fuzzware* and *MULTIFUZZ*, and Code Coverage of *AidFuzzer*

Appendix G.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

G.1. Summary

This paper presents *FIDO* (Fuzzing Input Delivery Optimizer), an extension for emulation-based firmware fuzzing frameworks that improves the delivery of fuzzing inputs in interrupt-driven embedded firmware. The approach models firmware input handling as "CRP input routes" consisting of availability Checks, data Retrieval, and Processing steps. FIDO uses a combination of static and dynamic analysis to identify appropriate delivery points and infer input-length bounds. Based on this model, FIDO schedules and distributes inputs across interrupt routes to avoid common issues such as data loss, starvation, or excessive input injection that limit existing fuzzers. The system is implemented as an extension for *Fuzzware*, *MULTIFUZZ*, and *SEmu* and evaluated on unit tests and 22 real-world firmware images, where it significantly improves code coverage and bug discovery, leading to the identification of several previously unknown vulnerabilities.

G.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

G.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. It addresses the problem of input delivery in interrupt-driven firmware fuzzing, demonstrating that the timing and quantity of injected inputs significantly influence coverage and bug discovery. By introducing a novel approach orthogonal to traditional improvements in test-case generation or execution backends, *FIDO* advances the state of the art.
2) The paper creates a new tool to enable future science. The authors implement FIDO as a practical extension compatible with existing firmware fuzzers such as *Fuzzware*, *MULTIFUZZ*, and *SEmu*, showing measurable improvements in coverage and vulnerability discovery. By planning to release *FIDO* as open source, the work provides the research community with a reusable tool that can support future studies and be combined with other fuzzing advancements.