SteamFuzz: Boosting Firmware Fuzzing via Automatic Message Window Inference (Extended Version)*

Jorik van Nielen
j.j.vannielen@utwente.nl
University of Twente
Enschede, The Netherlands

Andreas Peter
andreas.peter@uol.de
University of Oldenburg
Oldenburg, Germany

Andrea Continella
a.continella@utwente.nl
University of Twente
Enschede, The Netherlands

Abstract

With the increasing adoption of embedded devices across industries, concerns about their security are rising. While fuzzing has proved effective for detecting vulnerabilities in embedded devices, existing approaches treat inputs as "single-shot", whereas in reality firmware processes a stream of sequential messages (e.g., network packets). This lack of abstraction and message-awareness hinders the efficient exploration of deep, multi-message dependent code. Yet, leveraging message boundaries is challenging, since real-world protocols rely on variable-length, implicitly delimited messages, and firmware images lack explicit input format specifications.

To address this problem, we introduce SteamFuzz, a novel message-aware firmware fuzzing approach. SteamFuzz dynamically infers message windows for each newly discovered input before adding it to the corpus. Then, SteamFuzz leverages message windows to (1) drastically minimize inputs—increasing throughput and focusing mutations on meaningful bytes—and (2) perform message-level mutations—exploring message-dependent code.

We implement SteamFuzz and evaluate it on a diverse set of 40 firmware images, comparing its performance with state-of-the-art fuzzers. Our experimental results demonstrate that, thanks to the inferred message windows, SteamFuzz explores program behaviors more efficiently, reaching coverage faster and reaching code that existing tools miss, while generating significantly shorter inputs.

Keywords

Firmware Fuzzing, Rehosting, Software Security

1 Introduction

With the ever-increasing connectivity of modern society and the microcontroller market exceeding 52 billion USD in 2025 [20], embedded firmware is now present everywhere, from smart homes and vehicles to medical devices and critical infrastructure. As such, firmware security is of vital importance to guarantee confidentiality (e.g., smart home cameras), integrity (e.g., correct functioning of a pump), and availability (e.g., car sensors). Recent attacks demonstrate the importance of comprehensive firmware security evaluation [2, 33, 53]. However, the large variety of hardware and the opaque nature of embedded firmware [51, 56] make security analysis particularly challenging.

Recent advances in fuzz testing (fuzzing) embedded firmware have greatly improved our bug-finding capabilities. Instead of fuzzing devices with the hardware in the loop, state-of-the-art approaches "rehost" firmware, allowing for scalability and better performance [16]. Rehosting is then combined with MMIO (memory-mapped input/output) modeling to effectively simulate peripherals and inject (mutated) inputs [15, 18, 42, 60]. Recent advances further improve performance by treating inputs on a per-stream basis, essentially considering each MMIO address as a distinct input source [10, 44].

Yet, modern fuzzing tools treat firmware as *single-shot programs*—i.e., programs that, for each input source, receive and process only one input per execution. In reality, firmware is a continuously running program that receives and processes *messages*: structured sequences of multiple inputs delivered over time. For instance, drones receive a sequence of Bluetooth packets that together form messages instructing the drone to take off, accelerate, rotate, and land. In practice, overlooking the structured, multi-message nature of firmware fundamentally limits fuzzing on two fronts. First, inputs become excessively long, as superfluous messages cannot be trimmed, hindering effective mutation and reducing fuzzer throughput. Second, mutations at the message level, commonly applied to other message-based targets to improve performance (e.g., shuffling, injecting, or duplicating individual messages), are not possible.

For traditional multi-message programs, message boundaries are naturally enforced by the underlying interface. For example, network protocol messages are naturally delimited by the network interface, and similarly Linux driver interaction is delimited by the system call interface. Fuzzers targeting these programs leverage the provided message boundaries to generate single- and multi-message inputs [6, 12, 13, 22, 25, 26, 31, 36, 38, 46–48, 59]. For embedded firmware, however, we lack information on message boundaries, as firmware receives raw input through MMIO stream reads and interrupts, without message delimiters or structural cues—besides, firmware binaries lack explicit input format specifications. As a result, existing fuzzers have no inherent mechanism to distinguish where one message ends and the next begins.

Approach. We propose SteamFuzz, a novel approach that enables faster and more effective firmware fuzzing through *message window* inference. Before storing inputs in the corpus, we dynamically investigate the impact of interrupt timings on the resulting execution trace. This knowledge is then used to infer message windows, recovering the boundaries of messages. To this end, we leverage the

*This is the extended version of our paper accepted at the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands. This version includes additional appendices not present in the conference proceedings.

key observation that foreground functions (i.e., ISRs) are responsible for receiving data from the peripherals, while background functions (i.e., task functions) further process the input after a full message has been received. We leverage message windows to (1) reduce both the noise in, and the size of, the inputs stored in the corpus, and (2) apply high-level, message-level mutations to aid fuzzing in reaching deep execution states.

Evaluation. We evaluate SteamFuzz on a diverse set of 40 firmware images from state-of-the-art benchmarks. Our evaluation shows that SteamFuzz breaks through the coverage plateaus of existing fuzzers while storing up to 14× fewer inputs, each executing up to 37× fewer basic blocks, reading up to 20× fewer MMIO values, and firing up to 114× fewer interrupts. Moreover, our approach discovers more bugs, and discovers them faster, than existing tools.

Contributions. Overall, we make the following contributions:

- We propose a novel dynamic analysis approach to reconstruct message windows from firmware inputs.
- We develop SteamFuzz, a new firmware fuzzer that leverages our message window inference to boost firmware fuzzing performance.
- We evaluate the performance of our approach on 40 firmware images and demonstrate that it outperforms existing tools.

In the spirit of open science, we release SteamFuzz and our dataset at https://github.com/utwente-scs/steamfuzz_artifact.

2 Background & Motivation

We provide the necessary background on firmware design and firmware fuzzing, and we motivate the need for an approach like SteamFuzz with an example based on a real-world firmware image.

2.1 MCU Firmware Design

Embedded microcontroller (MCU) firmware is the software that runs on small, cost-efficient, and power-constrained devices. To perform efficiently on such constrained hardware, firmware generally does not feature a fully-fledged Operating System (OS). Instead, developers actively and specifically pick only the necessary components, e.g., task scheduling functionality, peripheral drivers, and communication libraries. The result is a single binary, so-called *monolithic firmware* or Type II/III firmware [35], that includes all different components and data. Linux-based firmware security has been studied widely, as data flows between user applications, drivers, and the kernel are easily visible [1, 39, 52]. Monolithic firmware, by contrast, interweaves all these components into a single opaque image, making security analysis considerably more difficult.

This paper focuses exclusively on monolithic firmware, hereafter referred to simply as firmware.

While firmware images come as singular self-contained binaries, they still rely on a functional structure. The runtime structure can be split into high-priority foreground and low-priority background processes. Figure 1 visualizes the runtime structure and the interaction between firmware and the peripherals.

Foreground – Interrupt Service Routines. The Interrupt Service Routines (ISRs) are low-level, high-priority functions that use

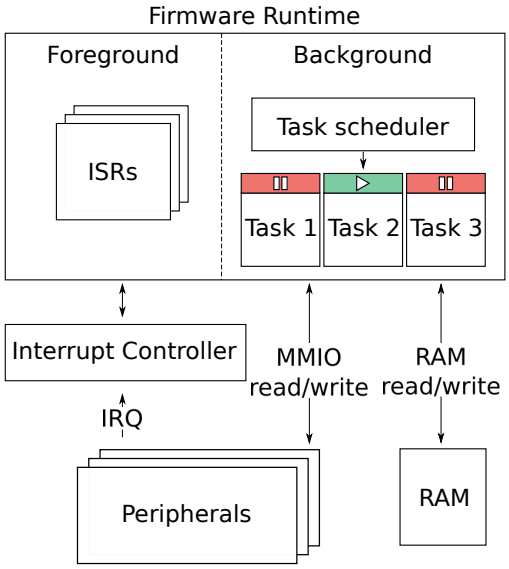


Figure 1: MCU firmware runtime consists of (a) foreground context, where high-priority ISRs handle incoming interrupts, and (b) background context, where tasks execute. The firmware interacts with the peripherals through the interrupt controller, and by reading/writing from MMIO addresses.

the Hardware Abstraction Layer (HAL) to communicate with the peripherals. The Interrupt Controller invokes an ISR when it receives a peripheral interrupt. On ARM Cortex-M devices, the Nested Vectored Interrupt Controller (NVIC) [3] determines whether to invoke an ISR, which ISR to invoke based on the IRQ number, and whether to preempt other interrupts based on interrupt priority. Before switching context to the ISR, the processor saves the current register values, including the program counter, to the stack. ISRs for data transfer peripherals commonly read incoming data from MMIO addresses and store it in shared data structures for further processing by background processes. Crucially, ISRs only process the incoming data to a limited extent, as they must complete quickly to prevent blocking other interrupts. ISRs return execution to background tasks by loading the `EXC_RETURN` value into the program counter (PC) register [4]. This special value signals the CPU to restore the background context by loading values stored on the stack back into their respective registers. Figure 2 shows a typical firmware execution trace (matching our motivating example introduced in Section 2.3). Several IRQs trigger ISR execution, which reads MMIO values (e.g., from memory ID 2) into a shared buffer (e.g., at memory ID 4).

Background – Looping Tasks. When no ISRs are active, infinitely running loops called *tasks* process incoming data. In the simplest firmware implementations, a single super loop performs all data processing. In more advanced firmware, often based on Real-Time Operating Systems (RTOSs) such as FreeRTOS [21] and RIOT [41], multiple tasks run according to a time-scheduling algorithm. Figure 2 shows how the task scheduler invokes tasks sequentially. ISRs preempt the background tasks, but upon ISR completion the

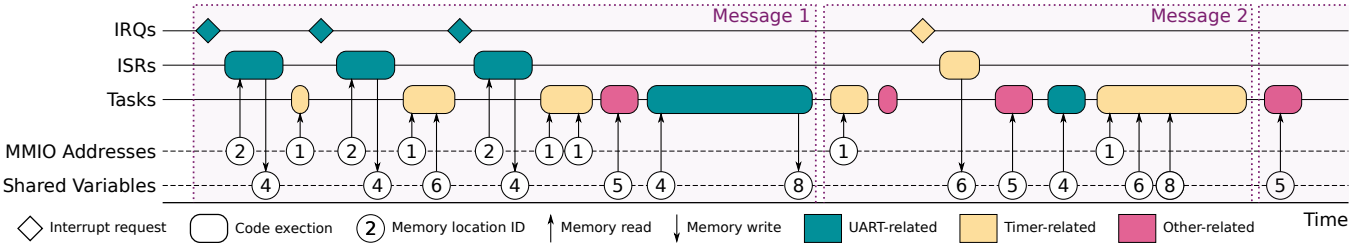


Figure 2: Illustration of firmware runtime. Interrupt requests (IRQs) cause the matching Interrupt Service Routine (ISR) to execute. ISRs are responsible for receiving new data from the peripherals by reading from MMIO addresses. After the ISR completes, control is handed back to the scheduler, which executes tasks that further process the data. Some tasks are idle and pass control back to the scheduler. Tasks might write to or read from shared variables. Messages are a sequence of interrupts and MMIO reads that result in the execution of a task. The colors of code execution match the functions in our motivating example (Figure 3).

interrupted task resumes. Tasks commonly read from shared variables populated by ISRs (e.g., memory IDs 4 and 6) or other tasks (e.g., memory ID 8). Notably, tasks also read MMIO values to check whether the peripheral is ready to receive data (e.g., memory ID 1). When idle, a task polls relevant MMIO addresses and shared variables to check for new data. For instance, the timer task in Figure 2 polls MMIO address 1 and shared variable 6 in message 1. If no new data is available, the task can either continue polling the same addresses (e.g., timer task) or yield control to the task scheduler, which then invokes another task (e.g., UART task). When new data arrives—either because an ISR moved it into a shared buffer (e.g., memory IDs 4 and 6) or another task updated a shared variable (e.g., memory ID 8)—the task will start processing the newly available data. For tasks that would otherwise yield, execution time increases, as visualized for the UART task in message 1.

2.2 Firmware Rehosting and Fuzzing

Rehosting-based approaches execute firmware in a virtual environment instead of on the microcontroller, enabling fuzzing of firmware at scale [16]. To facilitate the unique characteristics of firmware, rehosting-based fuzzers present inputs through three channels (Figure 1):

Memory-Mapped I/O. The microcontroller manages ranges of memory addresses that enable bidirectional communication with the peripherals. Each of these MMIO addresses maps to a peripheral register, and these registers provide diverse features. For instance, the control register manages peripheral behavior, the status register signals the firmware that new data is available, and the data register transfers newly received data to the firmware. State-of-the-art firmware fuzzers create models for each MMIO address [15, 42] and mutate inputs on a per-stream basis [10, 44].

Peripheral Interrupts. To enable the asynchronous nature of firmware, peripherals send interrupt requests (IRQs) to the microcontroller. One common use case for this is to notify the firmware that new data has been received by a peripheral. Upon receiving an IRQ, firmware invokes the corresponding Interrupt Service Routine (ISR).

Interrupts are handled by a dedicated hardware component called the *Interrupt Controller*. The Interrupt Controller is tightly integrated with the CPU and decides when the CPU switches context to an ISR. For ARM Cortex-M devices, the firmware configures the Interrupt Controller through a combination of the interrupt vector table and the NVIC. The interrupt vector table is an array of up to 256 ISR pointers. The first 16 entries are reserved for standard CPU interrupts, such as `Reset`, `BusFault`, and `SysTick`. The remaining 240 are configured based on the underlying MCU and used by peripherals to send IRQs to the CPU. The firmware can change interrupt vector table entries at runtime, enabling a different ISR to be triggered for the same IRQ. Interrupts are further configured through the NVIC data structure in four ways. First, the firmware can change the interrupt vector table location by writing to the `VectorTableOffsetRegister`. This allows the firmware to change all ISR locations at once, for example after the boot sequence completes. Second, the firmware can enable or disable interrupts on a per-IRQ basis. Third, the firmware can set priorities of each IRQ. When an interrupt is triggered during execution of an ISR, the Interrupt Controller interrupts the current ISR only if the priority of the new interrupt is higher. Fourth, all interrupts can be temporarily disabled by the firmware during sensitive code execution.

To follow and replicate such a mechanism, modern firmware fuzzers track the NVIC to list available interrupts, and fire IRQs both at regular intervals and when firmware is waiting for new inputs [10, 44, 55].

Direct Memory Access. Some peripherals use Direct Memory Access (DMA) to read and write data directly to and from the microcontroller's RAM. The firmware uses interrupts and MMIO communication to initialize and manage DMA data transfers.

For firmware fuzzers, configuring DMA buffers and determining when to copy data into them present significant challenges. The challenge of DMA is orthogonal to our work and addressed in prior research [32, 43]. Therefore, we do not model DMA data transfers.

2.3 Motivating Example

Figure 3 shows a simplified code snippet of a common peripheral communication pattern. The snippet represents firmware running on a motor-controlling MCU, and features two ISRs and two tasks. The UART peripheral sends an IRQ when a new byte is ready to be

```
1  struct ring_buffer uart_rx_buffer;
2  char command[MAX_COMMAND_SIZE];
3  struct motor_command pending_motor_command = {.type = MOTOR_NONE};
4  bool calibrated = false, control_tick = false;
5
6  void uart_rx_isr() {
7    ring_buffer_write(&uart_rx_buffer, UART->RXDATA);
8  }
9
10 void timer_isr() {
11   if (pending_motor_command.type != MOTOR_NONE) {
12     control_tick = true;
13   }
14 }
15
16 void uart_processing_task() {
17   while (1) {
18     if (is_empty(&uart_rx_buffer)) { task_yield(); continue; }
19     if (process_byte(next_byte(&uart_rx_buffer), command)) {
20       handle_command(command, &pending_motor_command);
21     }
22   }
23 }
24
25 void motor_control_task() {
26   while(1) {
27     if (MOTOR->STATUS == MOTOR_READY && control_tick) {
28       if (pending_motor_command.type == MOTOR_CALIBRATE) {
29         // ...
30         calibrated = true;
31       }
32       else if (pending_motor_command.type == MOTOR_SET_SPEED) {
33         change_motor_setting(&pending_motor_command);
34       }
35       control_tick = false;
36       pending_motor_command.type = MOTOR_NONE;
37     }
38   }
39 }
40
41 void handle_command(char *cmd, struct motor_command *out) {
42     if (strncmp(cmd, "CALIBRATE", 9) == 0) {
43       out->type = MOTOR_CALIBRATE;
44     }
45     else if (strncmp(cmd, "SET_SPEED", 9) == 0 && calibrated) {
46       out->speed = atoi(cmd + 10);
47       out->type = MOTOR_SET_SPEED;
48     }
49 }
```

Figure 3: A simplified firmware code snippet showing a UART and timer ISR (foreground) and the relevant tasks (background). The UART ISR reads each data byte into a ring buffer. The UART processing task reads and processes input from the ring buffer byte by byte; after a complete command is received, it is passed to the motor command handler. The timer ISR is triggered by a clock peripheral and sets a flag requesting a motor control update. The motor control task checks whether the motor is ready and a control tick is pending, then executes any queued command—either calibrating the motor or updating its speed. A calibration command must be executed before any speed command will be accepted.

read by the firmware. The interrupt controller invokes `uart_rx_isr` upon receiving the IRQ. The `uart_rx_isr` is a lightweight function that reads a byte from the relevant MMIO address (the DATA register), and writes it to the ring buffer. `uart_processing_task` is an infinitely looping function responsible for handling commands received over UART. The task scheduler regularly transfers control to this task. Whenever a new byte is available in the ring buffer, `process_byte` processes it. This function parses the input into a plaintext command, removing any headers. `process_byte` notifies the main loop once a whole command has been received by returning `true`. The command is then executed, potentially reaching new code coverage and altering the firmware state. Note that when no new byte is available, `uart_processing_task` transfers control back to the scheduler by calling `task_yield`.

The second ISR, `timer_isr`, is invoked when a timer expires, e.g., every 50ms. This ISR notifies the firmware when the motor settings should be updated. The `motor_control_task` is a second task function, responsible for controlling the motor. It continuously checks if the motor is ready to receive new settings (by reading the motor status register), if it is time to update the motor based on the `control_tick` variable, and whether a motor command is pending. If so, it updates the motor settings and resets both the `control_tick` and `pending_motor_command` variables.

Viewed from a high-level perspective, the input to this firmware consists of two distinct messages arriving from independent sources: a command arriving over UART, and a timer event triggered by a clock peripheral. Interaction with this firmware snippet is visualized in Figure 2.

This pattern is common in firmware: inputs arrive in sequences of messages of various types. Unfortunately, without properly recognizing message boundaries, this pattern hinders the ability of state-of-the-art fuzzers to achieve high coverage and deep bug detection, for two main reasons:

I. Long Inputs. Firmware fuzzing inputs often consist of long sequences of messages, many of which do not contribute to new coverage. In our example, a likely scenario is that multiple unimportant UART commands are received before an interesting command invokes new coverage. Without knowledge of message boundaries, a fuzzer cannot identify or remove these redundant messages from the input: trimming arbitrary bytes and interrupts mid-sequence risks corrupting the structure of subsequent messages, losing the ability to reach the new coverage. Consequently, newly generated inputs get longer as they are mutated from old ones, accumulating irrelevant messages.

The notorious downside of long inputs on fuzzers is two-fold. First, longer inputs take longer to execute, directly reducing fuzzer throughput. Second, the longer the input, the smaller the chance that a mutation lands on an interesting byte rather than on an irrelevant message. Together, these effects cause the fuzzer to spend more time per input execution, and to waste more executions on uninteresting inputs.

II. Lack of Message-Level Mutations. Some code paths are only reachable through a specific sequence of messages, yet without awareness of message boundaries, fuzzers cannot reason about or manipulate inputs at the message level. In our example, reaching line 46 requires a precise four-message sequence: a `CALIBRATE` UART command, a timer tick, a `SET_SPEED` UART command, and a second timer tick. A stream-level fuzzer does not mutate on the message level: it may stumble upon individual valid commands, but the probability of generating a correctly ordered, semantically coherent sequence of messages by mutating raw bytes is small. Without message-level mutations, the fuzzer is unlikely to systematically explore the interactions between messages, leaving portions of the firmware's code space unreachable.

2.4 Challenges

Effectively identifying message boundaries and grouping fuzzing inputs into messages faces three fundamental challenges.

I. Absence of universal message delimiters in input stream. Fuzzing inputs are stored as collections of streams: a sequence of bytes for each MMIO read address, and a separate stream for interrupts. Low-level communication protocols often use specific bytes to delimit a message, e.g., a UART interface may use a newline character to denote the end of a command. However, given the large variety of protocols and their different implementations, extracting this knowledge requires significant manual effort. Moreover, a delimiter byte does not always serve as a message boundary: consider a UART command that is followed by multiline arguments, where newlines appear mid-message. Interrupts are another candidate for message delimiters. However, protocols generally use multiple interrupts per message. For example, UART fires one interrupt for each individual incoming byte. Thus, solely relying on the analysis of fuzzing inputs is insufficient for accurate message inference.

II. Lack of runtime message processing indications. During firmware execution, tasks continuously poll for incoming data and process it when available. From the outside, these two states—processing a message and idly polling—are difficult to distinguish: in both cases the task executes, reads from buffers, and checks MMIO registers. Even when a task yields immediately upon finding no new data, one cannot trivially determine whether a message was processed in the preceding execution window. Without a reliable signal of when a task has finished processing a message, it is unclear where one message ends and the next begins.

III. Altering fuzzing inputs easily breaks message structure. Inferring message boundaries and performing message-level mutations both require careful manipulation of inputs to avoid corrupting the intended message structure. Without this care, modifications can have a cascading effect: altering one message causes later interrupts to read MMIO values intended for a different message, failing to trigger the intended behavior. For instance, removing a single interrupt without also removing the related MMIO values consumed by its ISR causes the next interrupt to consume values that were intended for the removed one, potentially derailing all subsequent execution.

3 Design

We present our threat model, followed by the design of SteamFuzz.

3.1 Threat Model

We adopt the same threat model as prior work in this field [10, 42, 44, 55]. The attacker has access to the firmware binary, but not to its source code. Additionally, the attacker knows the hardware the firmware runs on, specifically its memory map. At runtime, the attacker controls the data received through peripherals (via MMIO addresses) and the timing of interrupt firing.

3.2 High-Level Overview

We propose SteamFuzz, a novel approach to embedded firmware fuzzing featuring message window inference. Figure 4 shows an overview of SteamFuzz's execution loop, with our contributions highlighted in green. For the first iteration, we generate a random input and provide it to the firmware image under test while tracking the executed basic block edges. If new edge coverage is discovered, we infer message windows and minimize the input before storing it in the corpus. In subsequent iterations, we select a random seed from the corpus and mutate it at the byte, stream, and message level. The following sections detail the design of our main components: message window inference, input trimming, and message-level mutations.

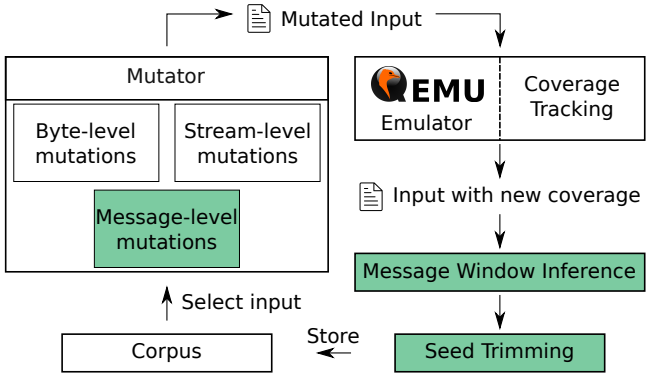


Figure 4: A high-level overview of SteamFuzz. Our contributions are highlighted in green.

3.3 Message Window Inference

Our key insight is that a complete message always culminates in task code execution: the task code processes the received data and updates the firmware state accordingly. Conversely, ISRs that do not trigger any data processing by task functions never mark the end of a message. These properties of task code and ISRs are inherent to embedded firmware (Section 2.1), and form the foundation of our generic methodology.

We leverage this insight to infer message windows in three steps, illustrated in steps 1–3 of Figure 5. Central to this process is the concept of the *interrupt interval*, i.e., the number of basic blocks executed between the return of the previous ISR and the triggering of the next interrupt. In short, we identify interrupts that require additional task-level processing beyond the ISR itself to locate message window boundaries.

1. Trailing Value Removal. After discovering an input that results in new coverage, we remove as many trailing input values (IRQs and MMIO-related) as possible from the end of the input while retaining the new coverage, following standard practice for rehosting-based firmware fuzzers [10, 44]. This effectively eliminates unimportant trailing messages that do not contribute to the new coverage.

2. Interrupt Tailing. *Tailed* interrupts are interrupts that fire directly after the previous ISR returns, without any background code execution in between. In essence, all interrupts in a chain of tailed interrupts have an interrupt interval of 0. Starting from the back of the input, we attempt to create a chain of as many tailed interrupts as possible while retaining the same coverage as the original input. This is an iterative process: first, we try to tail half of the interrupts in the input and verify whether the original coverage

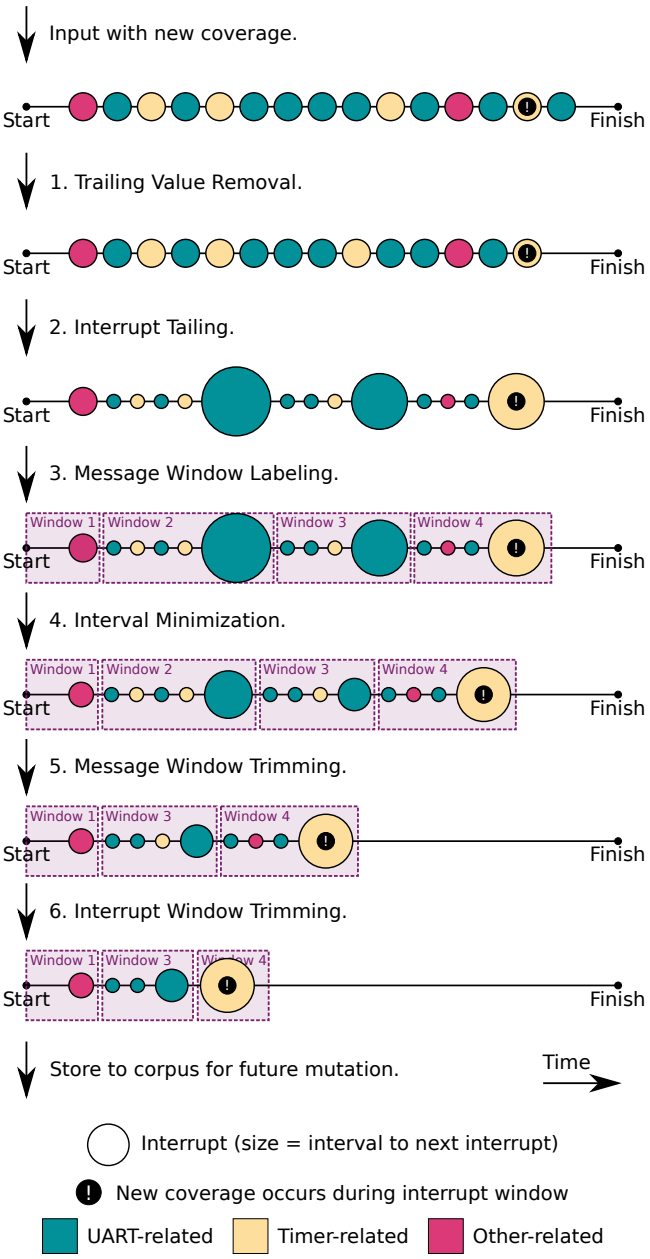


Figure 5: Message window inference. Inputs are shown in execution order from start to finish. Each circle represents an interrupt and the following MMIO bytes in the input. The size of the interrupt visualizes the interval between that interrupt and the next. The colors represent the IRQ number and match the ISRs in our example code snippet (Figure 3).

is still reached. We then proceed using binary search to find the maximum number of interrupts that can be tailed. To verify each candidate input, we execute it in the emulator.

After discovering the first tailed interrupt chain (at the end of the input), we know that the interrupt preceding it cannot be tailed. We therefore skip that interrupt and proceed by attempting to tail the interrupts before it, applying the same binary search approach. We continue iterating until we reach the first interrupt.

We tackle challenge III in two ways. First, we extend the interval of the final remaining interrupt by the sum of the reduced intervals, ensuring that task execution at the start of the next message follows the same basic blocks as in the original input. Second, to prevent unread MMIO values from spilling into the next message, we remove any MMIO values that were read in the original input at the end of the message but remain unread in the modified version.

3. Message Window Labeling. Interrupt tailing produces chains of tailed interrupts separated by interrupts with longer intervals. A longer interval indicates that tailing that specific interrupt prevented the input from reaching the same coverage. In other words, a longer interval indicates that task processing is required at that moment during execution. We have thus identified message windows: groups of interrupts that solely receive input bytes, terminated by an interrupt that is followed by task-level data processing. This approach addresses challenges I and II by combining insights from the input—are the interrupt intervals zero or greater?—and from runtime behavior—does the modified input still reach the same coverage?

We label each interrupt with a non-zero interval as the end of a message window, and the subsequent interrupt as the beginning of the next message. The start of the input is labeled as the start of the first message window, and the end of the input as the end of the last.

3.4 Seed Trimming

Seeds created by firmware fuzzers tend to grow rapidly in size. As discussed in Section 2.3, long inputs negatively impact performance. We therefore reduce the size of seeds before storing them in the corpus through a novel trimming approach, whose process is visualized in steps 4–6 of Figure 5.

4. Interval Minimization. We minimize each remaining non-zero interval one by one using binary search. For each candidate interval, we verify its validity by running the input in the emulator. To address challenge III, we remove any MMIO values that were read by the original message but not by the adjusted one, preventing them from spilling over into subsequent messages.

5. Message Window Trimming. We trim inputs at the message window level, retaining only those messages required to reach the new coverage. The last message window is always kept, as it is where the new coverage occurs following tail removal in step 1. We leverage Algorithm 1, which operates as a double loop. The outer loop iterates over the starting point of the candidate trimmed region, beginning at the first message window. The inner loop searches for the furthest ending point of the candidate trimmed region, starting from the window preceding the last. If removing the current range fails to preserve coverage, the inner loop retracts the ending point by one message. Once the inner loop identifies the maximum contiguous range to remove (possibly zero), the outer loop advances the starting point by one, since the first window in the remaining input is evidently required. After each removal, we run the trimmed input in the emulator to verify that the new coverage is still reached.

Algorithm 1 Window Trimming for both message and interrupt windows. RemoveWindow removes the leading interrupt and relevant MMIO bytes from the input streams. VerifyTrim runs the input in the emulator and ensures the new coverage is still reached.

```
1:  function TrimSeed(S)  ▷ S is a seed, consisting of multiple windows
2:      skip ← 0
3:      high_1 ← length(S) − 1
4:      while skip < high_1 do
5:          S' ← S
6:          high_2 ← high_1 − 1        ▷ Last window contains new coverage.
7:          found ← False
8:          while skip < high_2 do
9:              new_input ← RemoveWindow(S', skip, high_2)
10:             success ← VerifyTrim(new_input)
11:             if success then
12:                 found ← True
13:                 S ← new_input
14:                 break
15:             else
16:                 high_2 ← high_2 − 1
17:             end if
18:         end while
19:         if found then
20:             high_1 ← high_1 − (high_2 − skip + 1)
21:         end if
22:         skip ← skip + 1
23:     end while
24:     return S
25: end function
```

6. Interrupt Window Trimming. We further trim each remaining message window at the interrupt window level, where an interrupt window spans from one interrupt firing to the next, including all intermediate MMIO values. We reuse Algorithm 1 for this purpose, operating on interrupt windows in place of message windows.

3.5 Message-Level Mutations

Multi-message fuzzers for other domains widely apply message-level mutations to unveil deep code [5, 6, 9, 12, 13, 22, 25, 26, 31, 36, 38, 46–48, 50, 59]. Inspired by these works, SteamFuzz mutates inputs at the message level. Concretely, we insert one or more consecutive messages from another seed in the corpus. To address challenge III and prevent breaking the expected message structure, we check which interrupts are available at the insertion point and require the injected message to start with one of those interrupts.

4 Implementation

Following our design, we implement a proof-of-concept of SteamFuzz in about 1,600 lines of Rust on top of Hoedur [44]. We build our implementation for ARM Cortex-M firmware, as the platform is widely used and allows for direct comparison with the state-of-the-art. Our implementation is publicly available at https://github.com/utwente-scs/steamfuzz_artifact.

4.1 Fuzzing Platform

The emulation and low-level fuzzing setup of SteamFuzz closely resembles that of other embedded firmware fuzzers [10, 44, 55]. We build on QEMU-based emulation [7], with hooks into our fuzzer for events such as basic block execution and program crashes. We use the multi-stream input format introduced by Hoedur [44]. To improve performance, we take a snapshot during the first run at the point the first MMIO value is read, and start every subsequent run from this snapshot. We treat each executed edge as a feature, adding inputs that result in new features to the corpus. Edges resulting from context switches (e.g., task function to ISR) are discarded.

4.2 Interrupt Intervals

For our message inference approach, we require variable interrupt intervals. To this end, we equip each entry in the interrupt stream with an interval value. When the fuzzer triggers an interrupt, it consults the interrupt stream to retrieve both the IRQ number and the interval to the next interrupt. The interval denotes the number of basic blocks between the completion of the current ISR and the trigger of the next interrupt. An interval of 0 implies that the next interrupt immediately tails the previous: the return of the previous ISR is followed directly by the invocation of the next, without any instructions from the background context executing in between. We detect ISR completion via a QEMU hook that triggers when the PC is set to `EXC_RETURN`.

To support boot-required interrupts, we add a fuzzable input value controlling the delay before the first interrupt is triggered. We use a default interval of 1,000 basic blocks when generating new interrupts during mutation. When no interrupt is available at the moment an interval expires, we fall back to this default, repeating until an interrupt becomes available. Once the next interrupt is eventually triggered, the interval stored in the input is updated to reflect the actual number of basic blocks elapsed since the previous ISR returned.

4.3 Message Inference

Message inference requires careful adjustment of the fuzzer input. Since fuzzer inputs do not encode the order in which MMIO values are read, any adjustment requires re-executing the input to discover the actual read order. When tailing interrupts—setting all intervals to 0 except the final one—unread MMIO values can spill over into subsequent message windows. This occurs because tasks that originally executed during the intervals no longer run, leaving their MMIO values unread and available to the next message.

To prevent this, we run the minimized input up to the end of the targeted message window, cut off any unread MMIO values, and append the remaining messages from the original input. Similarly, for interval minimization, we run the input up to the point where the interrupt following the targeted interrupt would be triggered, cut off any remaining MMIO values in the targeted window, and append the original MMIO values that follow it. After each run, we verify that execution produced the expected intervals. For interval minimization we use binary search to find the shortest valid interval, though this does not always yield the global minimum: tasks frequently enable and disable interrupts dynamically, causing some shorter intervals to fail to invoke the correct interrupt. A linear scan is not feasible given that intervals can span tens of thousands of basic blocks, so we leave further minimization to the general fuzzing process. After executing a mutated input, we check whether

the number of MMIO values read or the number of basic blocks executed has decreased while covering the same edges as the seed; if so, we replace the original with the new input.

4.4 Message-level Mutations

In general, SteamFuzz applies multiple mutations to a seed before execution, following AFL's mutation stacking strategy [58]. Message-level mutations require knowledge of the processing order of the input streams, so we only apply them as the first mutation, since the execution order of each stored seed is known. This prevents us from having to execute an already-mutated input before knowing where to insert a new message window.

5 Evaluation

We evaluate the performance of our approach on state-of-the-art benchmarks and compare our tool to existing fuzzers. Specifically, we focus on addressing the following research questions:

- **RQ1:** How does input trimming affect the corpus?
- **RQ2:** How does SteamFuzz compare to the state-of-the-art in coverage?
- **RQ3:** How does SteamFuzz compare to the state-of-the-art in bug discovery?
- **RQ4:** How does the message window inference approach handle different low-level communication protocols?
- **RQ5:** How effective are message-level mutations and interrupt interval mutations?

5.1 Experimental Setup & Dataset

Experimental Setup. We run our experiments on a server equipped with two Intel Xeon 6527P CPUs (24 cores, 48 threads, 4.2 GHz) and 256 GB of RAM, running Ubuntu 24.04.4 LTS. In line with recommendations in the fuzzing literature [45], we run each target for 24 hours and repeat each experiment five times. To assess the performance of our approach with respect to the state-of-the-art, we compare SteamFuzz to Hoedur [44] and AidFuzzer [55]. Hoedur [44] is a state-of-the-art stream-based firmware fuzzer using Fuzzware [42] MMIO modeling. AidFuzzer [55] extends this with ISR modeling to trigger interrupts at relevant timings.

Dataset. We collect a diverse dataset to evaluate and compare our approach. We combine two existing datasets: (1) FirmBench [14], a benchmark dataset specifically designed for firmware fuzzers, containing 31 diverse samples featuring various RTOSs, SDKs, complex MMIO protocols, and network protocols; (2) AidFuzzer [55], a dataset of 10 samples that are highly dependent on correct interrupt timing. With one sample being present in both datasets (3DPrinter), we evaluate SteamFuzz on 40 firmware samples.

5.2 Input Management (RQ1)

Table 1 shows input seed metrics for SteamFuzz, Hoedur, and AidFuzzer after running each on our 40 firmware samples. Per sample, the highest coverage is shown in bold, as are the lowest (best) values for the remaining columns.

Overall, SteamFuzz effectively minimizes the number of inputs stored in the corpus while reaching similar coverage. For instance, for `Contiki_NG_Shell`, SteamFuzz finds similar coverage (1585/1587 basic blocks) while keeping 14× fewer inputs in the corpus compared to Hoedur. This tighter packing of coverage-triggering inputs in the corpus reduces noise. For the same sample, AidFuzzer stores 13× more inputs while only reaching similar coverage. Although AidFuzzer produces smaller corpora for some samples, the corresponding fuzzing performance is substantially lower than that of SteamFuzz. SteamFuzz stores significantly more inputs for five samples (e.g., `contiki-hello-4-4` and `PLC`). Upon further investigation, we find that SteamFuzz exploits an arbitrary code execution bug present in these samples, reaching new coverage freely and generating many inputs that each jump to a unique location.

The average input execution length is also significantly lower for SteamFuzz than for Hoedur: for the `nobcn_ffd_ncp` sample, the average is 37× shorter (7,600 basic blocks versus 283,900), while SteamFuzz achieves higher coverage. AidFuzzer similarly produces longer inputs for samples where it reaches the same coverage as SteamFuzz: for `contiki-router`, AidFuzzer covers 177 fewer basic blocks while averaging 181,300 basic blocks per input execution, compared to 10,300 for SteamFuzz (17×). Shorter inputs increase fuzzer throughput, as each execution requires fewer basic blocks to be processed.

The average number of MMIO reads per input is also lower for SteamFuzz: for the `zephyr-sam4s` sample, SteamFuzz produces 19.5× fewer MMIO reads per input than Hoedur. For another sample, `bcn_rfd_ncp`, AidFuzzer averages 10.7× more MMIO reads, while reaching 49% less coverage. This indicates that the trimming strategies work effectively, reducing the likelihood of the fuzzer mutating irrelevant bytes.

Inputs stored by SteamFuzz also contain fewer interrupts. For the `Contiki_NG_Shell` sample, all three fuzzers reach similar coverage, yet SteamFuzz averages only 3 interrupts per stored input, compared to 82 for Hoedur (27×) and 341 for AidFuzzer (114×). Additionally, AidFuzzer fails to generate any interrupts for some samples in the FirmBench dataset, as it is unable to produce effective interrupt models for those samples, which reduces its overall performance.

Overall, compared to the state of the art, SteamFuzz achieves similar or greater coverage with substantially more efficient input management: it stores fewer inputs, each containing fewer MMIO values, and triggers fewer interrupts and basic block executions.

5.3 Coverage (RQ2)

We compare the coverage performance of SteamFuzz against Hoedur and AidFuzzer. We run all three fuzzers on our full dataset of 40 firmware samples, with the aggregated results shown in Figure 6. Run-to-run comparisons are shown in the Appendix (Figure 9).

For various samples (e.g., `bcn_rfd_ncp`, `blehci`, and `Zephyr_SocketCan`), SteamFuzz manages to find coverage never reached by the other tools. These results confirm that our message window inference and input trimming techniques improve the fuzzing exploration and effectiveness.

The `3Dprinter` sample shows how our message inference significantly speeds up fuzzing. While both SteamFuzz and Hoedur reach approximately the same coverage in 24 hours, SteamFuzz

Firmware sample	Coverage (BBs)			Inputs in Corpus			Avg BBs / Input (x1000)			Avg MMIO Reads / Input			Avg IRQs (Msgs) / Input		
	Hoedur	AidFuzzer	SteamFuzz	Hoedur	AidFuzzer	SteamFuzz	Hoedur	AidFuzzer	SteamFuzz	Hoedur	AidFuzzer	SteamFuzz	Hoedur	AidFuzzer	SteamFuzz
annepro-shine	837	813	**1015**	1363	873	**184**	74.3	**12.4**	73.7	615	819	**82**	73	210	**37 (37)**
bcn_rfd_ncp	351	1200	**2353**	1975	**392**	537	91.7	21.8	**13.6**	1991	4789	**449**	90	82	**25 (4)**
blehci	898	720	**1808**	252	**111**	270	13.2	**5.2**	7.4	290	319	**28**	80	67	**8 (2)**
coord_ncp	1736	1854	**2027**	762	737	**190**	273.0	**20.8**	30.9	3985	3002	**1139**	268	111	**5 (5)**
mac_no_beacon_sleep	793	**1940**	1916	**154**	1296	575	**6.4**	20.9	36.4	**1065**	4089	3653	**0**	61	13 (10)
nmea	1948	1908	**2002**	148	118	**43**	50.3	64.0	**10.5**	333	9976	**80**	43	16	**14 (2)**
nobcn_ffd_ncp	2045	1226	**2155**	1691	423	**254**	283.9	9.7	**7.6**	4038	1350	**316**	281	34	**8 (2)**
sam4l_qtouch	**566**	565	**566**	108	138	**16**	193.9	65.7	**35.6**	526	2419	**271**	192	676	**70 (48)**
taulab	**3226**	1858	3130	2064	424	**251**	70.7	**9.4**	28.7	863	640	**435**	53	64	**2 (1)**
3Dprinter	4775	4478	**4809**	5212	2984	**1370**	155.5	53.9	**36.8**	2320	4667	**1758**	151	579	**24 (3)**
6LoWPAN_Receiver	**3482**	2482	3445	2459	893	**304**	200.6	**43.6**	49.8	7659	3032	**972**	192	117	**24 (2)**
betaflight	**483**	—	**483**	38	—	**9**	**25.6**	—	27.8	**1024**	—	2058	**0**	—	**0 (0)**
CNC	**2845**	2615	2843	6320	2377	**919**	94.7	64.8	**14.8**	2421	5021	**137**	93	107	**2 (2)**
Console	**796**	784	794	340	211	**34**	69.6	20.8	**4.8**	141	735	**26**	69	40	**8 (3)**
contiki-6lowpan	1704	1692	**2013**	1691	**976**	21481	78.9	83.1	**8.2**	1401	3008	**343**	74	250	**3 (3)**
contiki-hello-4-4	2254	1156	**2979**	2141	**419**	24238	87.7	64.5	**10.0**	3176	1901	**151**	78	227	**6 (6)**
contiki-hello-4-8	2002	1158	**2821**	1467	454	**377**	103.9	82.6	**16.6**	1612	2339	**255**	99	268	**5 (5)**
contiki-router	2435	2309	**2486**	2263	1244	**358**	88.5	181.3	**10.3**	2583	3531	**243**	82	186	**2 (2)**
contiki-snmp	1459	1196	**1470**	861	502	**121**	65.7	160.9	**10.6**	828	2448	**129**	58	230	**2 (2)**
Contiki_NG_Shell	**1587**	1564	1585	773	702	**53**	87.0	153.7	**18.2**	1025	3049	**190**	82	341	**3 (2)**
Gateway	3277	2957	**3280**	3558	**2249**	5683	148.6	**138.8**	145.5	624	918	**190**	136	131	**24 (6)**
GPSTracker	1014	—	**1047**	1033	—	**175**	135.0	—	**31.7**	603	—	**80**	134	—	**8 (3)**
hoverboard	1486	560	**1506**	1723	**127**	217	221.4	128.7	**102.4**	1058	17143	**162**	192	1538	**9 (3)**
loramac	**2521**	1639	2485	709	**79**	80	266.4	**12.8**	59.5	3678	494	**484**	36	**0**	4 (4)
oresat-control	**2910**	—	1455	2901	—	**165**	296.5	—	**100.5**	1375	—	**68**	199	—	**7 (7)**
PLC	1944	357	**2154**	5666	**61**	19208	115.4	14.0	**6.6**	446	815	**77**	115	70	**16 (2)**
RF_Door_lock	737	17	**777**	—	**3**	24	—	**15.0**	30.0	—	**17**	30	—	**0**	**0 (0)**
RIOT_CCN_LITE	**2143**	—	2140	618	—	**71**	130.7	—	**83.9**	450	—	**134**	83	—	**14 (3)**
RIOT_GNRC	**997**	181	912	531	**9**	41	63.7	**0.2**	33.3	921	76	**13**	88	**0**	2 (2)
riot_gnrc_networking	5188	120	**7048**	2963	**20**	859	254.6	**0.4**	156.6	2192	**217**	363	157	**0**	6 (4)
Soldering_Iron	3315	—	**3469**	**9914**	—	26491	63.8	—	**15.0**	2357	—	**283**	60	—	**7 (3)**
utasker_MODBUS	**1624**	—	1595	785	—	**102**	140.2	—	**25.6**	631	—	**323**	79	—	**5 (2)**
utasker_USB	2074	—	**2705**	**17073**	—	19482	139.1	—	**110.9**	560	—	**48**	21	—	**1 (1)**
zephyr-3330	3008	1702	**3037**	878	**68**	216	100.6	37.5	**26.2**	2292	788	**220**	149	**0**	4 (4)
zephyr-bt	1075	483	**1096**	546	**5**	118	36.9	16.8	**5.9**	588	207	**58**	149	**0**	13 (6)
zephyr-f429zi	**1273**	530	1271	605	**17**	84	37.0	**8.9**	13.6	829	90	**54**	32	**0**	1 (1)
zephyr-nrf	**2302**	942	1175	792	356	**121**	36.7	**4.7**	6.5	3332	345	**58**	108	**0**	1 (1)
zephyr-sam4s	**3154**	1749	3112	1300	**79**	213	126.0	23.1	**22.3**	3929	1192	**201**	226	**0**	3 (3)
zephyr-sampro	3458	1838	**3461**	1612	**108**	266	96.5	**28.0**	42.7	1976	1735	**467**	177	**0**	3 (3)
Zephyr_SocketCan	3122	847	**3284**	2385	**37**	316	361.8	**19.0**	77.0	1150	313	**69**	207	**0**	3 (2)

Table 1: Corpus comparison between Hoedur, AidFuzzer, and SteamFuzz. For each firmware sample, we select the median run by coverage. We report corpus size and the average number of basic blocks, MMIO reads, and IRQ triggers per input execution for that run, considering only scheduled (non-crashing) inputs. For SteamFuzz we also include the average number of message windows. The highest coverage value and lowest values for all other metrics are highlighted per sample. AidFuzzer was not able to run on seven samples (e.g., GPSTracker), as it got stuck in an early crashing state.

reaches the plateau in less than 8 hours, while Hoedur takes around 24 hours—resulting in a 3× speed-up.

We manually investigate the three samples for which Hoedur significantly outperforms SteamFuzz. For all three samples (`RIOT_GNRC`, `oresat-control`, and `zephyr-nrf`), the firmware exhibits a significant gap between short and long message lengths, which SteamFuzz struggles to bridge. We discuss this limitation in detail in Section 6.

5.4 Bug Finding Ability (RQ3)

To evaluate bug finding ability, we run SteamFuzz, Hoedur, and AidFuzzer on the 31 firmware targets from the FirmBench dataset. We use the FirmReBugger bug oracles (called ravens) [14] to accurately detect bugs, and exclude ravens used for false positive crashes. We implement the bug oracle for both SteamFuzz and AidFuzzer, and use the provided implementation for Hoedur. Figure 7 shows the bug survival rates of the three fuzzers. We only show results for the 15 samples where at least one fuzzer fails to find all bugs within the first hour.

For 7 samples (`contiki-hello-4-8`, `contiki-snmp`, `riot_gnrc_networking`, `utasker_USB`, `zephyr-sam4s`, `zephyr-sampro`, and `Zephyr_SocketCan`), SteamFuzz finds more bugs and/or finds them faster. For `Zephyr_SocketCan`, SteamFuzz triggers bugs that require precisely crafted CAN messages. Through reduced noise in the stored inputs, SteamFuzz finds one bug that Hoedur does not. The same holds for `riot_gnrc_networking`: SteamFuzz carefully crafts UART messages and finds crashes that Hoedur misses. For 5 samples (`6LoWPAN_Receiver`, `Gateway`, `GPSTracker`, `utasker_MODBUS`, and `zephyr-f429zi`), both fuzzers find bugs at similar timings. For the remaining 3 samples, Hoedur outperforms SteamFuzz. For `contiki-hello-4-4`, SteamFuzz exploits a discovered arbitrary code execution bug to reach new coverage, inadvertently distracting the fuzzer from finding other bugs. For `zephyr-nrf`, SteamFuzz fails to reach the coverage

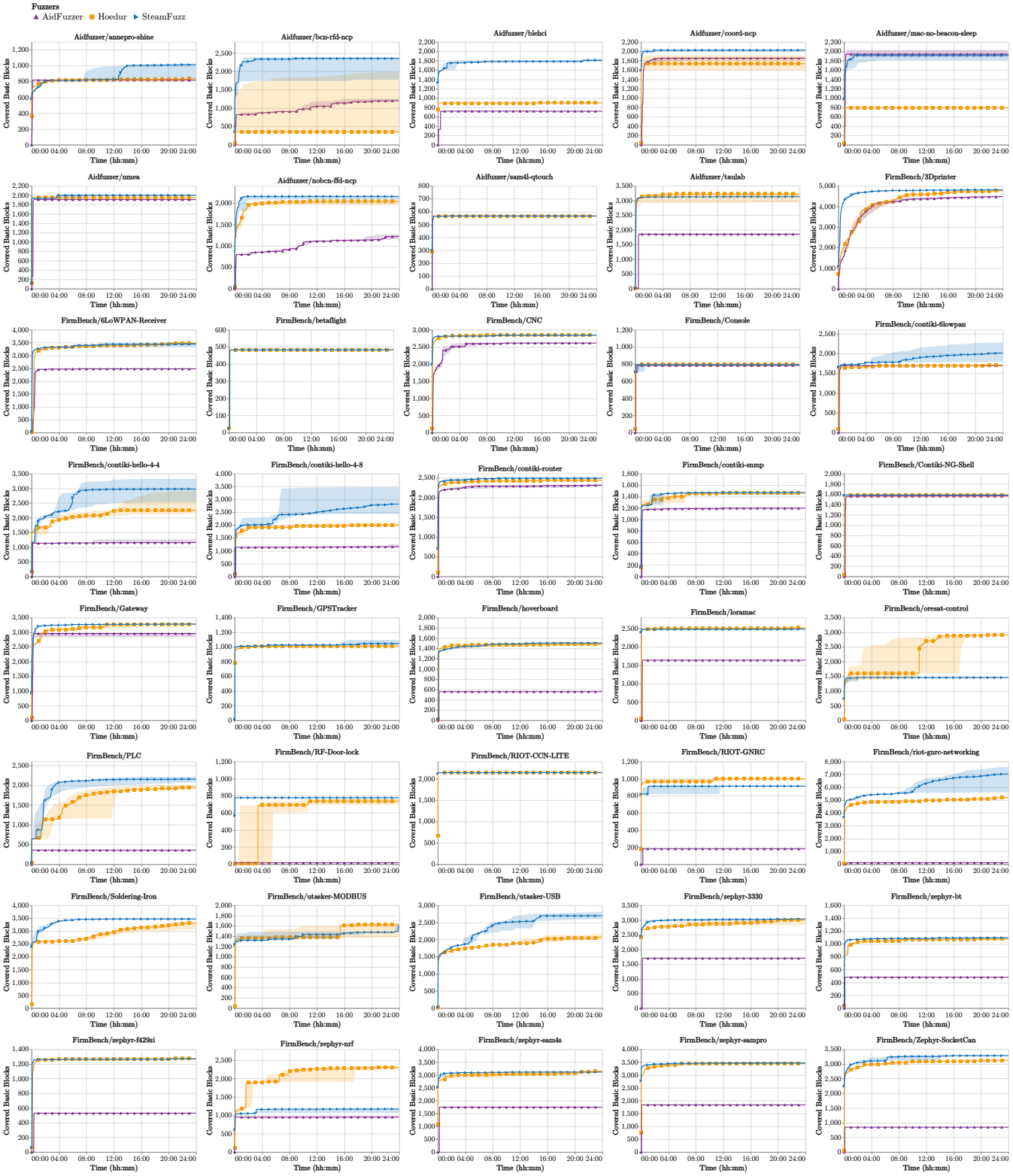


Figure 6: Comparison of the fuzzing performance of SteamFuzz, AidFuzzer, and Hoedur. For each firmware sample, we plot the coverage over time. The median and 66% confidence interval are shown. For 17 samples, SteamFuzz achieves coverage not reached by the other fuzzers, while for 4 samples it falls short of the coverage achieved by Hoedur. AidFuzzer was not able to run on 7 samples (e.g., `betaflight`), as it got stuck in an early crashing state.

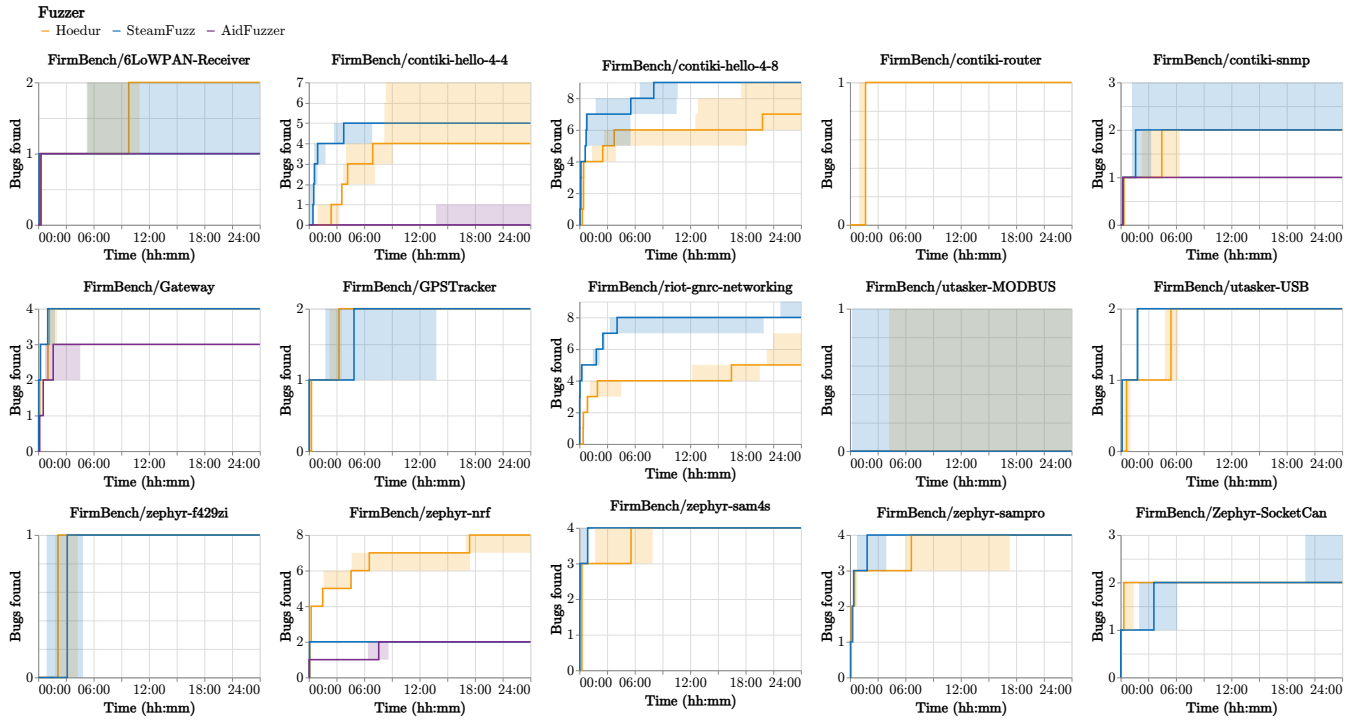


Figure 7: Bug survival plot showing the time to discover bugs per sample. Samples where Hoedur and SteamFuzz find all bugs within one hour are excluded, as these do not distinguish the approaches. The median and 66% confidence interval are shown.

required to trigger the bugs, as discussed in Section 5.3. Finally, for `contiki-router`, SteamFuzz reaches the relevant code, but fails to trigger the bug, as doing so requires repeatedly invoking the same function within a single message—a pattern SteamFuzz discourages by trimming redundant loop iterations. We discuss this limitation in Section 6.

Newly Discovered Bugs. While the samples in our dataset have been heavily scrutinized by other works, we discovered two new out-of-bounds write vulnerabilities in the `utasker_MODBUS` and `blehci` firmware samples. We reported both vulnerabilities to their respective upstream projects. Both reports were confirmed, and the vulnerabilities were patched.

5.5 Message Inference (RQ4)

We investigate how our message inference approach handles inputs for the different firmware samples relying on different protocols. Unfortunately, no prior work has established ground truth for message windows in firmware inputs. Moreover, deriving exact message window boundaries manually requires extensive reverse engineering and execution path tracing.

To discover if our approach successfully addresses the three challenges listed in Section 2.4, we manually investigate whether SteamFuzz's inferred messages match the expected structure of the various underlying low-level communication protocols. Table 1 shows the average number of messages and interrupts in the last column.

UART/USART. The Universal Asynchronous Receiver-Transmitter (UART) and the Universal Synchronous / Asynchronous Receiver-Transmitter (USART) protocols transfer data byte by byte. A simplified firmware implementation using UART is shown in Figure 3. At the interrupt level, two core implementations exist.

In the first implementation, the firmware receives an interrupt for each newly available byte. This is reflected by SteamFuzz: samples using this approach (e.g., `3Dprinter`, `bcn_rfd_ncp`, `RIOT_CCN_LITE`, `hoverboard`) produce messages consisting of many consecutive tailed interrupts. As shown in Table 1, inputs for `3Dprinter` contain an average of 3 messages and 24 interrupts, corresponding to an average message size of 8 interrupts. We find most messages consist of complete GCode commands (e.g., a first `T0` message to select tool 0, followed by an `M104 S0` message to turn off the heater).

In the second implementation, the firmware receives a single interrupt and then reads bytes in a loop, polling the status register for new data. This is also observed in practice: for samples using this approach (`zephyr-nrf`, `Zephyr_SocketCan`, `annepro-shine`), SteamFuzz produces single-interrupt messages, with all bytes read within one interrupt handler.

SPI. The Serial Peripheral Interface (SPI) leverages a main-secondary architecture for synchronous communication. When firmware reads data over SPI, it keeps reading bytes until the peripheral does not have bytes available. As such, SteamFuzz finds single-interrupt messages for the samples featuring SPI (e.g., `coord_ncp`, `loramac`, `zephyr-nrf`). Investigating the `coord_ncp`

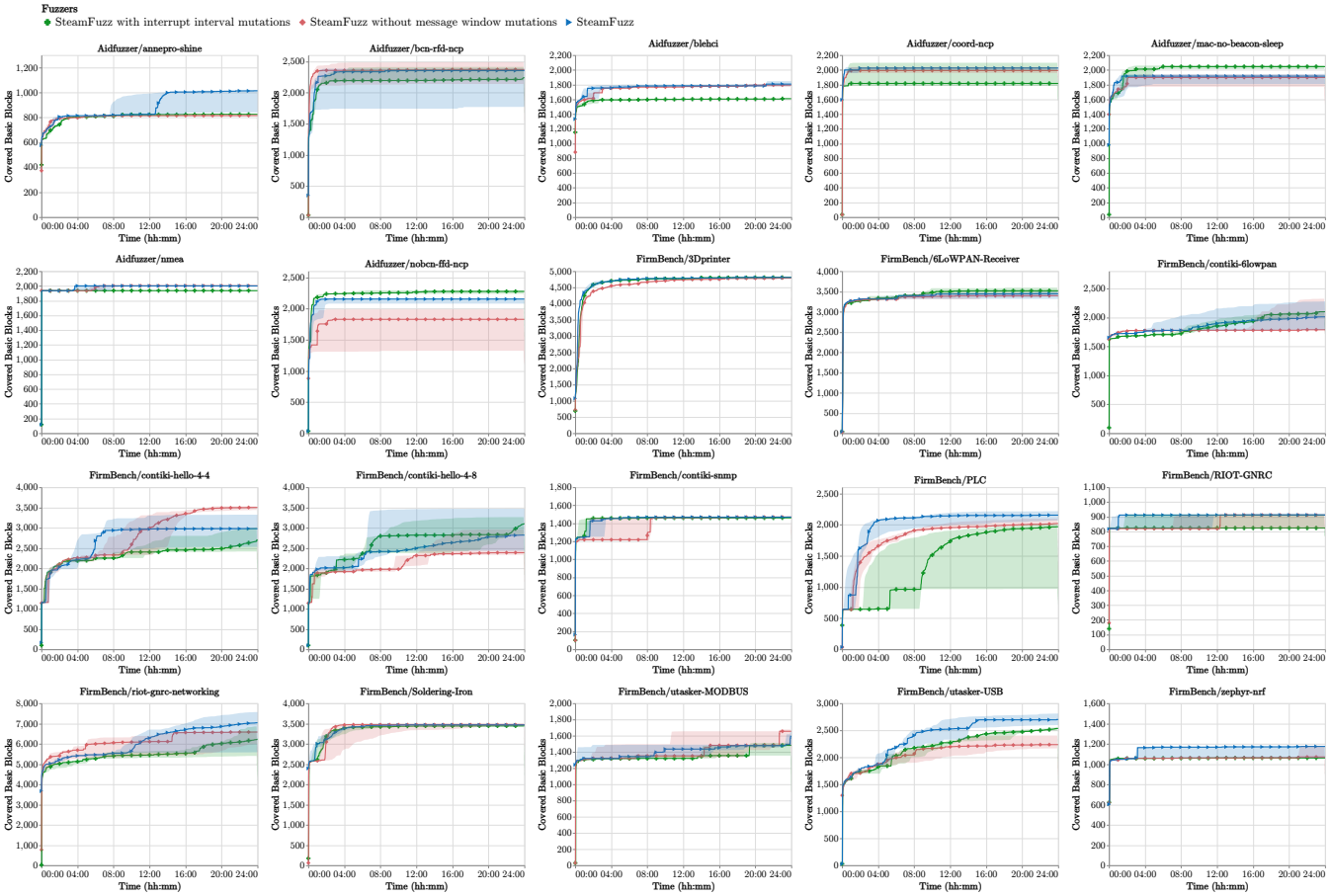


Figure 8: Ablation study of SteamFuzz. We compare SteamFuzz to SteamFuzz without message windows, and SteamFuzz plus interrupt interval mutations. The median and 66% confidence interval are shown.

sample, we find that the messages consist of reading one IEEE 802.15.4 packet buffer and adding it to a queue. A dedicated task then processes the message by parsing the binary data. The average input in the corpus contains 5 messages.

USB. The Universal Serial Bus (USB) protocol sends messages in packets. The firmware reads incoming data from several peripheral registers. For the only sample featuring USB in our dataset, `utasker_USB`, SteamFuzz produces single-interrupt messages for the USB messages.

Timers. Most firmware features a clock or timers, whose interrupts increment counter variables. As some events only occur after count variables pass certain thresholds, SteamFuzz produces clock messages consisting of multiple tailed interrupts (e.g., `utasker_MODBUS`, `GPSTracker`).

CAN. The Controller Area Network (CAN) protocol is a networked message bus, where frames use IDs to address the receiver. When the firmware receives a CAN message, it obtains the full frame by reading from multiple peripheral registers at once. For the sample featuring CAN (`Zephyr_SocketCan`), SteamFuzz produces single-interrupt messages, in line with expectations.

5.6 Ablation Study (RQ5)

We investigate the impact of message-level mutations on SteamFuzz by comparing two fuzzing configurations: (1) SteamFuzz *without message-level mutations*—message window inference and trimming are enabled, but message-level mutations are disabled; (2) SteamFuzz—our full approach. Additionally, we introduce a third configuration with an extra mutator that operates on interrupt intervals: SteamFuzz *plus interrupt interval mutations*—our full approach, extended with interrupt interval mutations.

Prior work has demonstrated that interrupt timing plays a key role in reaching new coverage [55]. We therefore include two complementary mutation strategies: one that reduces interrupt intervals, targeting early interrupts required during the boot process and helping shrink corpus input sizes; and one that increases interrupt intervals, targeting the dynamic interrupt scheduling patterns that firmware commonly exhibits as it enables and disables interrupts at runtime.

We run all three configurations on our dataset of 40 samples. Figure 8 shows how the different configurations perform for the 20 samples with the largest impact. The other 20 samples, for which the impact is marginal, are shown in the Appendix (Figure 10).

SteamFuzz without message-level mutations performs slightly better in early stages of, among others, `riot_gnrc_networking` and `bcn_rfd_ncp`. This makes sense, as in early fuzzing stages, sequences of messages are not yet semantically coherent. However, we also observe that in the longer term, SteamFuzz almost always outperforms the others.

The addition of interrupt interval mutations shows an advantage in two of the tested samples: `mac_no_beacon_sleep` and `nobcn_ffd_ncp`. We find that the addition of interval mutations results in the discovery of interesting interrupt timings. However, interrupt interval mutations also significantly reduce coverage for other samples, including `blehci` and `PLC`. We find that this is due to early interrupt intervals being significantly shortened, which reduces the ability to grow larger inputs. In other words, it worsens the Message Size Gap problem discussed in Section 6.

6 Limitations & Future Work

While SteamFuzz demonstrates a significant performance increase over the state-of-the-art, we identify two main limitations.

I. Message Size Gaps. SteamFuzz aggressively trims inputs, resulting in fewer and shorter inputs in the corpus, boosting fuzzing performance. However, for some samples the highly trimmed inputs result in challenges in bridging *message size gaps*. For instance, if a firmware sample communicating over UART has one command of 2 bytes and another command of 10 bytes, SteamFuzz struggles to reach the second message. Non-trimming fuzzers save inputs with more UART bytes to the corpus, even though those bytes are unimportant for the current coverage. They do, however, enable the discovery of longer commands through mutations. SteamFuzz struggles to bridge these gaps, as the chance of mutating a short input into a much larger one is small.

II. Hit-Count Bucketing. Modern coverage-guided fuzzers (AFL [58], LibFuzzer [37], Hoedur [44]) use Hit-Count Bucketing to effectively fuzz targets containing loops. Rather than recording only which edges were hit, the bitmap entry is also determined by how often each edge was hit. As a result, inputs that iterate a loop more often will also be seen as new coverage, and thus result in the input being stored in the corpus. For message-level fuzzers, hit-count bucketing should be applied on a per-message level. The current design of SteamFuzz does not allow for this, as at the time we execute a mutated input we are unaware of the message windows—we only infer message windows after we find that an input results in new coverage. As such, SteamFuzz can struggle to reach code only reachable through many loop iterations.

We identify these two limitations as the cause of SteamFuzz's decreased performance compared to Hoedur in 3 out of 40 samples. Naturally, both fuzzers can be deployed in parallel to maximize coverage and leverage the strengths of both approaches.

The message awareness provided by SteamFuzz can support future efforts to further improve firmware fuzzing. For instance, monitoring inter-message state could introduce an additional corpus dimension beyond edge coverage [59]. Furthermore, analyzing dependencies between messages could help generate more effective and interesting inputs [31].

7 Related Work

7.1 Embedded Firmware Fuzzing

General coverage-guided greybox fuzzing was popularized by the success of AFL [58] and later LibFuzzer [37]. Ever since, fuzzers have discovered countless bugs in real-world software.

Embedded firmware fuzzing is a widely explored topic. One of the fundamental challenges is handling peripheral device behavior: what data is provided through MMIO addresses and what interrupts should be triggered, and when. *Peripheral proxying*-based approaches run the firmware in an emulator while proxying peripheral interaction to the real device [24, 28, 29, 34, 57]. While effective, these approaches require physical access to the hardware, limiting scalability.

Instead of proxying real peripherals, *high-level emulation* replaces HAL-level functions in the firmware with stubs that read inputs directly rather than relying on MMIO values [11, 30, 49], though this often requires manually reverse-engineering HAL functions to determine the types of inputs they expect. To eliminate the need for any manual effort for firmware fuzzing, a range of works address hardware-free MMIO model generation through peripheral documentation processing [61], pattern-based approaches [18], or symbolic execution [27, 42, 60]. Recent work introduced multi-stream inputs to add structure to fuzzer inputs and generate them on a per-MMIO basis [10, 44], greatly increasing fuzzing performance. AIM proposes just-in-time interrupt firing through symbolic execution of ISRs [17]. Similarly, AidFuzzer models ISRs and their impact on global state to manage interrupt firing [55].

SteamFuzz advances the field by treating firmware as a multi-message program, adding structure to fuzzer inputs by inferring message windows.

7.2 Message-Level Fuzzing

Many domains have been addressed through message-level fuzzing. For most, message boundaries are determined by the underlying interface: network protocol fuzzers rely on frames [6, 13, 25, 36, 38], kernels and drivers rely on system calls [12, 22, 59], hypervisors rely on hypercalls [26, 46, 47], virtual devices rely on device I/O requests [31], full systems rely on network packets [48], and Trusted Execution Environments rely on API calls [9, 50]. IJON [5] is a domain-neutral state-aware fuzzer that requires manual annotation to denote message boundaries within the program.

Firmware inputs lack boundaries introduced by the underlying interface. Instead, SteamFuzz automatically infers message boundaries from the firmware execution, requiring no manual effort or prior knowledge of the communication protocol.

7.3 Seed Trimming

Traditional fuzzers such as AFL [58], AFL++ [19], and LibFuzzer [37] trim corpus seeds by testing whether ranges of the input can be removed without affecting the coverage reached. Single-stream firmware fuzzers like Fuzzware [42] and Ember-IO [15] rely on the same approach, as they delegate corpus management to these traditional fuzzers. Hoedur [44] and MultiFuzz [10] additionally apply trimming at the input stream level. By leveraging inferred

message windows, SteamFuzz trims inputs at the stream, interrupt window, and message window level.

7.4 Static Analysis

Various static analysis techniques identify components in firmware binaries. ArgXtract derives security configurations [51], FrameD identifies frameworks [54], Heapster finds and analyzes dynamic allocator functions [23], Hestia locates RTOS tasks [8], and FirmXray pinpoints and analyzes Bluetooth-related functions [56]. These approaches rely on compiler-, library-, or RTOS-specific patterns and are consequently subject to both false positives and false negatives. To infer message windows, SteamFuzz instead leverages dynamic analysis and the foreground-background split, which holds regardless of the compiler, library, or RTOS used.

8 Conclusions

We introduced SteamFuzz, the first fuzzer to infer message windows for embedded firmware. Through an iterative process of adjusting interrupt intervals and tracing the resulting executions, SteamFuzz infers message boundaries within unstructured firmware inputs. We leverage this knowledge to trim inputs, reducing noise in the corpus, and to apply targeted mutations at the message level. Our evaluation on state-of-the-art benchmarks shows that SteamFuzz outperforms existing fuzzers, reaching higher coverage plateaus and triggering bugs faster.

The introduction of message windows opens two directions for future research. First, our technique allows approaches developed for other message-based systems to be applied to embedded firmware. Second, we lay a foundation for state-aware fuzzing of embedded firmware.

Acknowledgments

We thank our reviewers for their valuable comments and inputs. This work has been supported in part by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project "FirmPatch", the HIVA project (Vidi TTW No. 22249) financed by the Dutch Research Council (NWO) under the grant https://doi.org/10.61686/LMSYB97788, the German Federal Ministry for Economic Affairs and Energy (BMWE) under agreement no. 03EI4101D (TRACEY), and the joint programme "zukunft.niedersachsen" of the Ministry for Science and Culture of Lower Saxony and the Volkswagen Foundation in the TransVer project.

References

[1] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. 2023. FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules. In *Proceedings of the USENIX Security Symposium.*

[2] Ionut Arghire. 2024. Unpatched Sceiner Smart Lock Vulnerabilities Allow Hackers to Open Doors. https://www.securityweek.com/unpatched-sceiner-smart-lock-vulnerabilities-allow-hackers-to-open-doors/ Accessed: April 27, 2026.

[3] Arm Ltd. 2006. Nested Vectored Interrupt Controller (NVIC) — ARM Architecture Reference Manual. https://developer.arm.com/documentation/ddi0337/e/Nested-Vectored-Interrupt-Controller Accessed: April 27, 2026.

[4] Arm Ltd. 2023. EXC_RETURN — Exceptions and Interrupts Overview. https://developer.arm.com/documentation/107706/0100/Exceptions-and-interrupts-overview/EXC-RETURN Accessed: April 27, 2026.

[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*

[6] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *Proceedings of the USENIX Security Symposium.*

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Security Symposium.*

[8] Constantin Brinza, Peng Liu, Jorik van Nielen, Daan Prinsze, Marius Muench, and Andrea Continella. 2026. HESTIA: Automated Application Code Identification in RTOS Firmware. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).*

[9] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. 2023. TEEzz: Fuzzing trusted applications on COTS Android devices. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*

[10] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. 2024. MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware. In *Proceedings of the USENIX Security Symposium.*

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

[12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[13] Cristian Daniele, Timme Bethe, Marcello Maugeri, Andrea Continella, and Erik Poll. 2025. LibAFLstar: Fast and State-Aware Protocol Fuzzing. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[14] Mathew Duong, Michael Chesser, Guy Farrelly, Surya Nepal, and Damith C Ranasinghe. 2026. FirmReBugger: A Benchmark Framework for Monolithic Firmware Fuzzers. In *Proceedings of the USENIX Security Symposium.*

[15] Guy Farrelly, Michael Chesser, and Damith C. Ranasinghe. 2023. Ember-IO: Effective Firmware Fuzzing with Model-Free Memory Mapped IO. In *Proceedings of the ACM ASIA Conference on Computer and Communications Security (ASIA CCS).*

[16] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. 2021. SoK: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the ACM ASIA Conference on Computer and Communications Security (ASIA CCS).*

[17] Bo Feng, Meng Luo, Changming Liu, Long Lu, and Engin Kirda. 2023. AIM: Automatic interrupt modeling for dynamic firmware analysis. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2023), 3866–3882.

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

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

[20] Fortune Business Insights. 2026. Microcontroller Market Size, Share & Industry Analysis, By Product Type (8-bit, 16-bit, and 32-bit Microcontroller), By Application (Networking & Communication, Automotive, Consumer Electronics, Industrial, Medical Devices, and Military & Defense) and Regional Forecast, 2026–2034. https://www.fortunebusinessinsights.com/microcontroller-market-106430 Accessed: April 27, 2026.

[21] FreeRTOS 2026. *FreeRTOS.* https://www.freertos.org/index.html Accessed: April 27, 2026.

[22] Google. 2026. syzkaller: Unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller Accessed: April 27, 2026.

[23] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. 2022. Heapster: Analyzing the security of dynamic allocators for monolithic firmware images. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*

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

[25] Ahmad Hazimeh, Duo Xu, Qiang Liu, Yan Wang, and Mathias Payer. 2024. Tango: Extracting Higher-Order Feedback through State Inference. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*

[26] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. 2017. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*

[27] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted firmware rehosting for embedded systems. In *Proceedings of the USENIX Security Symposium.*

[28] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. PROSPECT: Peripheral proxying supported embedded code testing. In *Proceedings of the ACM ASIA Conference on Computer and Communications Security (ASIA CCS).*

[29] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*.

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

[31] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. 2023. ViDeZZo: Dependency-aware virtual device fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

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

[33] Midnight Blue. 2026. Have you tried turning it off and on again? On bricking OT devices (part 1). https://www.midnightblue.nl/blog/have-you-tried-turning-it-off-and-on-again-part-1 Accessed: April 27, 2026.

[34] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar 2: A multi-target orchestration platform. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*.

[35] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[36] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A greybox fuzzer for network protocols. In *Proceedings of the IEEE International Conference on Software Testing, Validation and Verification (ICST)*.

[37] LLVM Project. 2026. LibFuzzer — a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html Accessed: April 27, 2026.

[38] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. NSFuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–26.

[39] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[40] Dennis Reidsma, Jeroen van der Ham, and Andrea Continella. 2023. Operationalizing cybersecurity research ethics review: From principles and guidelines to practice. In *Proceedings of the International Workshop on Ethics in Computer Security (EthiCS)*.

[41] RIOT 2026. *RIOT*. https://www.riot-os.org/ Accessed: April 27, 2026.

[42] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *Proceedings of the USENIX Security Symposium*.

[43] Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, Simon Wörner, Daniel Klischies, Felix Buchmann, Nils Ole Tippenhauer, Thorsten Holz, and Marius Muench. 2025. GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays. In *Proceedings of the USENIX Security Symposium*.

[44] Tobias Scharnowski, Simon Wörner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2023. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *Proceedings of the USENIX Security Symposium*.

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

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

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

[48] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network fuzzing with incremental snapshots. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*.

[49] Lukas Seidel, Dominik Christian Maier, and Marius Muench. 2023. Forming Faster Firmware Fuzzers. In *Proceedings of the USENIX Security Symposium*.

[50] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. 2021. RTKaller: State-aware task generation for RTOS fuzzing. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–22.

[51] Pallavi Sivakumaran and Jorge Blasco. 2021. argXtract: Deriving IoT security configurations via automated static analysis of stripped ARM Cortex-M binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[52] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, et al. 2023. Greenhouse: Single-service rehosting of Linux-based firmware binaries in user-space emulation. In *Proceedings of the USENIX Security Symposium*.

[53] Ken Tindell. 2023. CAN Injection: Keyless Car Theft. https://kentindell.github.io/2023/04/03/can-injection/ Accessed: April 27, 2026.

[54] Jorik van Nielen, Andreas Peter, and Andrea Continella. 2024. FrameD: Toward automated identification of embedded frameworks in firmware images. In *Workshop on the Security of Industrial Control Systems & of Cyber-Physical Systems (CyberICPS)*.

[55] Jianqiang Wang, Qinying Wang, Tobias Scharnowski, Li Shi, Simon Woerner, and Thorsten Holz. 2025. AidFuzzer: Adaptive Interrupt-Driven Firmware Fuzzing via Run-Time State Recognition. In *Proceedings of the USENIX Security Symposium*.

[56] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXray: Detecting Bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

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

[58] Michal Zalewski. 2017. American fuzzy lop.

[59] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *Proceedings of the USENIX Security Symposium*.

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

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

A Open Science

SteamFuzz together with our entire experimental setup is available at https://github.com/utwente-scs/steamfuzz_artifact.

B Ethical Considerations

All experiments in this work were conducted exclusively on publicly available firmware drawn from widely used fuzzer benchmark datasets [14, 55]. None of the analyzed firmware images are proprietary or deployed in production systems, so any bugs surfaced by SteamFuzz affect only the benchmark images themselves rather than end users. Our tool is designed with a clear defensive purpose: to improve the security of embedded firmware through more effective vulnerability discovery, enabling vendors and maintainers to identify and patch issues before they can be exploited in the wild.

We follow coordinated vulnerability disclosure practices [40] and reported the two vulnerabilities we discovered to the affected vendors prior to publication, allowing a reasonable remediation window before details are made public.

We further note that SteamFuzz itself is a defensive instrument rather than an offensive one. It does not generate weaponized exploits, and its outputs are oriented toward triage and patching rather than exploitation. While we acknowledge that, in principle, any improvement in fuzzing technology could be repurposed by malicious actors, we believe the security benefits of proactively surfacing and patching firmware vulnerabilities substantially outweigh this residual risk. We therefore see SteamFuzz as a contribution that strengthens the broader ecosystem of firmware security.

C Appendix



Figure 9: Run-to-run comparison of the fuzzing performance of SteamFuzz, AidFuzzer, and Hoedur. For each firmware sample, we plot the coverage over time.

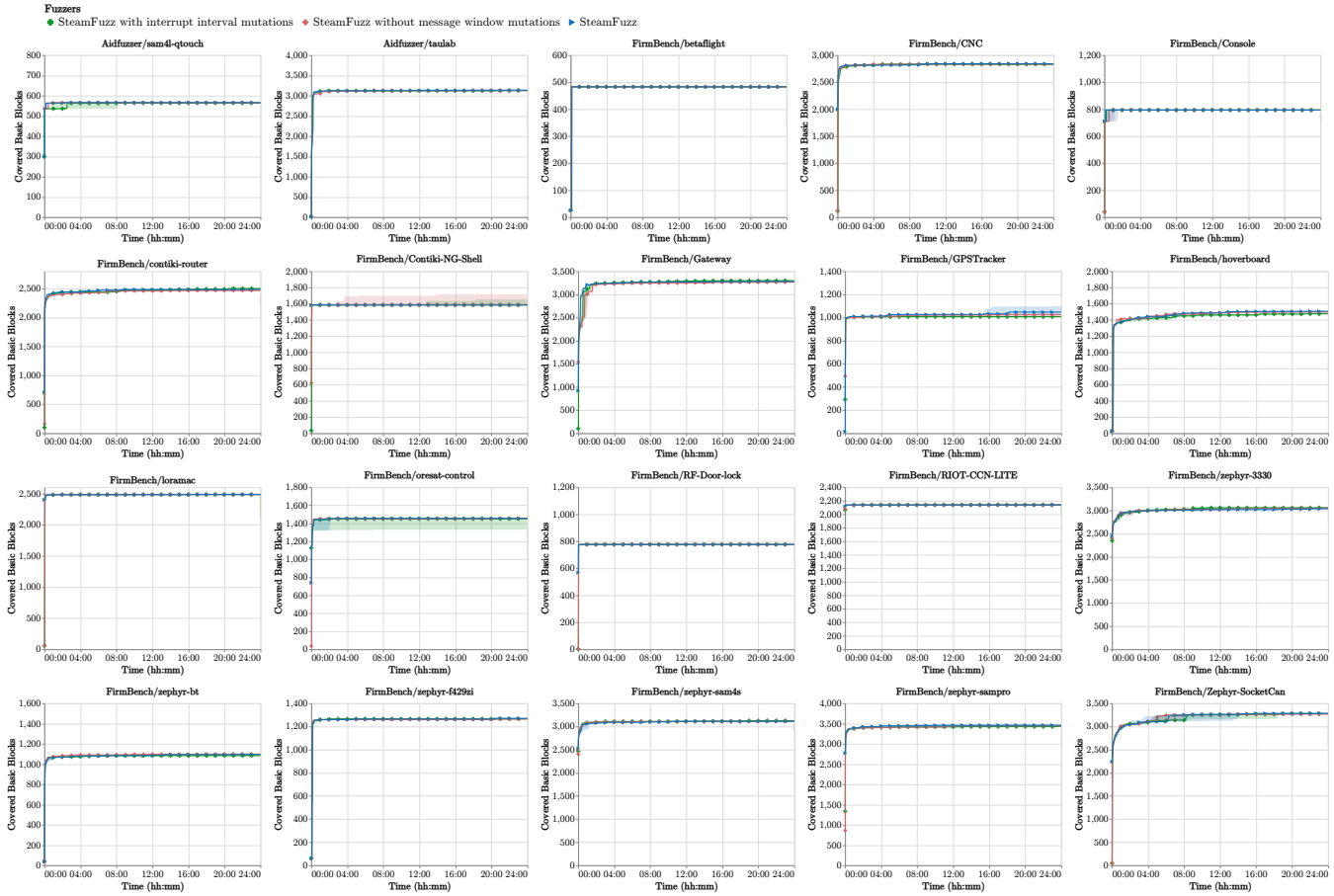


Figure 10: Ablation study of SteamFuzz, with samples for which differences are minimal. We compare SteamFuzz to SteamFuzz without message windows, and SteamFuzz plus interrupt interval mutations. The median and 66% confidence interval are shown.