HOEDUR: Embedded Firmware Fuzzing using Multi-Stream Inputs

Tobias Scharnowski and Simon Wörner, *CISPA Helmholtz Center for Information Security;* Felix Buchmann, *Ruhr University Bochum;* Nils Bars, Moritz Schloegel, and Thorsten Holz, *CISPA Helmholtz Center for Information Security*

https://www.usenix.org/conference/usenixsecurity23/presentation/scharnowski

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

HOEDUR: Embedded Firmware Fuzzing using Multi-Stream Inputs

Tobias Scharnowski*, Simon Wörner*, Felix Buchmann‡, Nils Bars,
Moritz Schloegel, and Thorsten Holz

CISPA Helmholtz Center for Information Security
‡*Ruhr University Bochum*

Abstract

Embedded systems with their diverse, interconnected components form the backbone of our digital infrastructure. Despite their importance, analyzing their security in a scalable way has remained elusive and challenging. Recent firmware rehosting work has brought scalable, dynamic analyses to embedded systems, making fuzzing for automated vulnerability assessments feasible. As these works focus on modeling device behavior rather than fuzzing, they integrate with off-the-shelf fuzzers in an ad-hoc manner. They re-interpret traditional flat binary fuzzing input as a sequence of hardware responses. In practice, this presents the fuzzer with an input layout that is *fragile*, *opaque*, and *hard to mutate* effectively.

Our work is based on the insight that while firmware emulation recently matured significantly, the input space is presented to the fuzzer in an ineffective manner. We propose a novel method for a firmware-aware fuzzing integration based on multi-stream inputs. We reorganize the previously flat, sequential, and opaque firmware fuzzing input into multiple strictly typed and cohesive streams. This allows our fuzzer, HOEDUR, to perform type-aware mutations and maintain its progress. It also enables firmware fuzzing to use state-of-the-art mutation techniques. Overall, we find that these techniques significantly increase fuzzing effectiveness. Our evaluation shows that HOEDUR achieves up to 5x the coverage of state-of-the-art firmware fuzzers, finds bugs that other fuzzers do not, and discovers known bugs up to 550x faster. In total, HOEDUR uncovered 23 previously unknown bugs.

1 Introduction

Embedded systems are ubiquitous, interconnected, and at the heart of smart homes, critical infrastructure, as well as the Internet of Things. From a security perspective, they are challenging to analyze because they are diverse, i. e., there is no common architecture and different vendors use custom hard- and software to build embedded systems. Moreover, these systems continuously process inputs from multiple hardware peripherals, such as serial data, sensor readings, and over-the-air packets. This poses significant challenges to testing the security of these systems. One particularly promising approach to addressing these challenges is *automated firmware rehosting*, which allows different firmware to run in generic emulators [13, 18, 28, 47, 60]. Recent advances in this area have enabled dynamic analysis techniques to scale [55].

Some of these firmware rehosting approaches apply fuzz testing (*fuzzing*) [13,18,47,60], a dynamic analysis technique that sends random inputs to the system under test. However, these works mainly focus on modeling hardware peripheral behavior to successfully run a given firmware image in an emulator. They do not specifically study the firmware fuzz testing process itself. Instead, they integrate off-the-shelf fuzzers into their system in an ad-hoc manner. In existing work, the emulator naïvely translates fuzzer-provided bytes sequentially in a *flat* manner [18,47,60]. Every time the firmware accesses one of its peripherals, the next few fuzzer-provided bytes are interpreted dynamically to serve the request.

We identify two inherent issues with this ad-hoc integration of existing fuzzers: First and most importantly, this *flat* processing of fuzzer-generated input causes the input structure to be *fragile* due to its sequential nature; even the smallest changes to early input bytes may change the firmware execution path and, thus, may also change the interpretation of the following fuzzing input bytes in an avalanche effect. This is due to the fact that firmware continuously processes data from multiple hardware sources. To respond to data retrieved from its different peripherals (e. g., a temperature sensor), the firmware may interrupt its current processing to react to the changes in its environment. Interrupts are crucial to react to external events in a timely fashion. Thus, the execution flow of the firmware directly depends on the fuzzer-generated data. Due to this frequent switching of firmware logic between its tasks, the data of each task is *scattered* across the input. For example, firmware may interrupt the retrieval of an IPv6 packet to read a sensor value. This interruption consumes data, such that the IPv6 packet is split within the fuzzer-generated input.

[*] Authors contributed equally to this work

This, in turn, renders many existing mutation techniques ineffective, such as multi-byte arithmetic or dictionary-based insertions, as they assume continuous data. Consequently, mutations frequently discard fuzzing progress because previously well-structured inputs are interpreted differently. A second downside of the ad-hoc integration of general-purpose fuzzers is that the fuzzer does not account for embedded firmware-specific behavior. For example, firmware is designed to run for an infinite amount of time, while general-purpose applications typically process a finite input.

In this work, we devise a *firmware-aware* fuzzer that utilizes a multi-stream input representation. As opposed to previous work, which represents hardware behavior as a *flat* binary input, we subdivide firmware inputs into multiple data streams, each representing a distinct use of a hardware register. This results in a well-structured input format containing strongly-typed data. Our multi-stream approach has multiple advantages: First, we can mutate the data pertaining to one particular hardware feature (one data stream) without impacting existing semantics of another hardware feature (another data stream). In contrast to a flat binary input format, this *stabilizes* our inputs, allowing the fuzzer to *preserve progress*. Second, our strongly-typed data streams allow us to perform *type-aware mutations*. Third, we are now able to collect per-stream performance metrics, which the fuzzer can use to *prioritize* mutating state-rich parts of the input.

One challenging aspect of our approach is the sheer amount of data streams: As firmware typically accesses a high number of hardware registers, we require hundreds of streams. Manually identifying and configuring them is therefore infeasible. We solve this by automatically assigning a data stream for each unique hardware access that we observe during emulation. A second challenge lies in the fact that most hardware registers represent the rather uninteresting low-level status of a device as opposed to actual application data. This makes the majority of data streams undesirable to mutate extensively past a certain point. To focus on mutating interesting data over time, we utilize a size-weighted version of Thompson sampling to schedule data streams for mutation.

We implement our approach in a prototype called HOEDUR. We evaluate HOEDUR on a sample set of 32 firmware images used in the evaluation of previous work. Our evaluation shows that HOEDUR achieves up to 5x the coverage of comparable state-of-the-art firmware fuzzers and triggers known bugs up to 550 times faster. HOEDUR also uncovered 23 previously unknown bugs, including 8 new bugs in the firmware images contained in the data set of previous work. We have responsibly disclosed these issues to their respective vendors in a coordinated way. So far, 22 CVEs have been assigned.

In summary, we make the following key contributions:

- We propose a novel multi-stream firmware input representation that allows firmware fuzzers to effectively mutate input and preserve fuzzing progress.
- We present the design and implementation of HOEDUR, an efficient firmware-aware fuzzer utilizing our multi-stream input representation.
- In a comprehensive evaluation, we show that our approach allows HOEDUR to significantly outperform state-of-the-art fuzzers in firmware fuzzing. As part of the evaluation, HOEDUR uncovered 23 novel bugs, which we responsibly disclosed to their respective vendors.

To foster research on this topic, we will release HOEDUR and the experimental data set at https://github.com/fuzzware-fuzzer/hoedur.

2 Firmware Fuzzing

Before introducing our multi-stream firmware input representation, we provide a brief overview of firmware fuzzing and challenges in this area.

2.1 Embedded Systems Firmware

Embedded systems are usually purpose-built and can consist of a number of different peripheral devices. The software running on these systems is known as *firmware*. Due to size and resource constraints, firmware is often built using one of the many specialized operating systems. So-called *bare-metal firmware* may also implement the required functionality itself, without an operating system. The firmware is responsible for all functions provided by the embedded system, ranging from the business logic down to the driver communicating with its peripheral devices. Logical tasks within embedded firmware are usually strongly coupled with an operating system library. This contrasts general-purpose software, where tasks are isolated and interact only with the operating system, using a set of system calls. Firmware with strongly-coupled tasks is also known as *monolithic firmware*.

2.2 Firmware-Peripheral Communication

To communicate with its surrounding hardware peripherals, firmware drivers use a handful of mechanisms for different purposes, e. g., Memory-Mapped Input/Output (MMIO) to modify and/or retrieve the state of a peripheral. Each peripheral has a memory region within the physical address space assigned to it. Within this memory region, the peripheral exposes a set of *MMIO registers*. Store operations to these registers allow the firmware to modify the peripheral state (such as setting its configuration), while load operations allow the firmware to retrieve the peripheral state or read actual data. As a given system typically contains a variety of peripherals, which in turn contain multiple MMIO registers that again are comprised of different bit fields, firmware typically interacts with its hardware in hundreds to thousands of distinct ways.

In addition to the firmware-initiated MMIO, peripherals can notify the firmware of asynchronous events, such as the arrival of new data, via an interrupt request (IRQ). An IRQ is a signal to the CPU raised by either a hardware peripheral or the firmware. The CPU will suspend its current task and switch to the corresponding *interrupt service routine* (ISR) to process the request.

As a third means for peripheral device communication, some embedded systems also use *direct memory access* (DMA). As the name implies, this allows the peripheral to read and write physical memory without CPU involvement. This works asynchronously to the firmware executing code on the microcontroller unit (MCU), is used for high-throughput devices, and is coordinated via MMIO and interrupts.

2.3 Rehosting-based Firmware Fuzzing

Firmware Rehosting describes the process of executing a firmware image in a virtual environment outside of its original physical device [55]. Successfully rehosting a firmware image requires the environment to mimic the behavior of the different physical hardware peripherals. More specifically, the rehosting environment needs to provide the communication mechanisms introduced previously, i. e., it needs to generate values for loads from MMIO registers, raise interrupts, and fill DMA input buffers with data.

As previous work has demonstrated, it is possible to provide a thin emulation layer and use fuzzer-generated input to approximate the peripheral device behavior for dynamic analysis [18, 47, 60]. This approach is appealing for broad testing, as this generic type of rehosting does not require specialized hardware or target-specific emulators and, thus, scales well with general-purpose computation resources.

Recent rehosting environments [18, 47, 60] allow a fuzzer to approximate peripheral device behavior by consuming flat fuzzing input in a streaming manner. Whenever the firmware communicates with its peripherals, e. g., by loading an MMIO register or reading from a fresh DMA buffer, the emulator draws the next required number of bytes from the fuzzer's binary input file. The emulator interprets these next bytes as the peripheral's response. This makes state-of-the-art fuzzers straightforward to integrate, as the emulator can act towards the fuzzer as an ordinary application. Like a command-line program, the emulator accepts a flat binary input file, reads from it, and consumes its contents piece by piece to sequentially respond to peripheral accesses.

3 Firmware Input Representation

To motivate why the current approach of processing fuzzer input falls short, we first need to discuss how firmware processes the input from its peripherals. We then analyze how the ad-hoc integration via flat inputs translates into firmware behavior and how this flat input representation impacts the fuzzer's ability to apply meaningful mutations and explore firmware behavior effectively. We then identify underlying areas of improvement that guide our design of multi-stream firmware fuzzing.

```
int main() {
    while ( true ) {
        Temperature::manage_heater();
        GCodeQueue::advance();
    }
}

void isr_usart() {
    /* Check data availability */
    if(usart_MMIO->SR & USART_SR_RXNE) {
        /* Feed char into GCode ringbuf */
        rb_push_insert(rb, usart_MMIO->DR);
    }
}

void isr_timer() {
    /* Periodic check: printhead height */
    if ( gpio_MMIO->IDR & ZFLAG) {
        /* Handle printhead height trigger */
        Planner::endstop_triggered(Z_AXIS);
    }
}
```

Figure 1: Simplified source code of the Marlin 3D printer firmware [16]. The `main` function handles heating and processes 3D printing GCode commands. `isr_usart` provides GCode data to `main`, while `isr_timer` periodically checks and manages the height of the printhead.

3.1 Firmware Input Processing

During its ordinary operation, the firmware runs in its main execution context. The main context may consist of a single loop or may be split into several tasks, each of which processes its own type of hardware input. Whenever a peripheral indicates an event by raising an IRQ, the main execution is suspended, and an ISR is executed to process the interrupt. After handling all pending interrupts, the firmware resumes execution in its main context. During its execution in different contexts, the firmware continuously accesses MMIO registers to interact with its surrounding peripherals.

For our analysis and as a running example, we describe a simplified version of the popular open-source 3D printer firmware Marlin [16]. At its core, the firmware accepts serial input in the form of *GCode*, which contains printing instructions. The firmware translates these instructions into actions for its printhead. During operation, the firmware must also track the movements of the printhead. Figure 1 outlines the Marlin source code in a simplified form. The featured code contains three execution contexts: the main loop (`main`) as well as two interrupt handlers (`isr_usart` and `isr_timer`). The main loop manages heating and processes GCode instructions. `isr_usart` adds the required input characters to the GCode ring buffer, and `isr_timer` periodically checks movement limits via a GPIO-connected magnetic sensor. In case a limit is reached, the Marlin firmware executes a function to

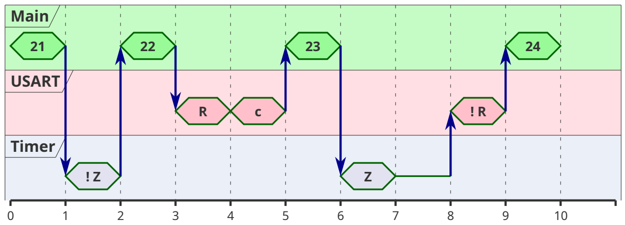


Figure 2: Illustration of hardware accesses in a 3D printer firmware grouped by execution context (its main loop and two ISRs). Each MMIO read is represented by one box; values are colored by context. Transitions between tasks caused by interrupts are shown as arrows. The values `R`, `c` and `Z` indicate the USART status register data availability flag, a USART data byte, and the Z-axis GPIO sensor reading, respectively. `!R` and `!Z` represent inversions.

handle the condition. Figure 2 visualizes how the firmware accesses its peripheral MMIO registers over time from different execution contexts. As we can see, the firmware logic switches between its different contexts, interleaving multiple peripheral accesses over time.

3.2 Processing Flat Inputs

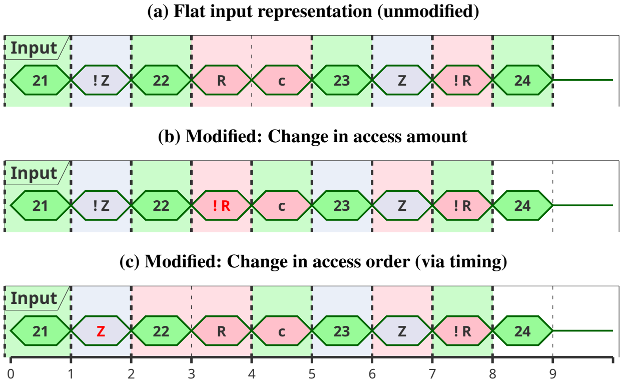
Given this example firmware, which interleaves its peripheral accesses, we now discuss how a flat, sequential input is translated into responses from these peripherals. We then analyze how a flat representation leaves the input *unstable* under common mutations, i. e., how mutating a given part of the input may change the meaning of its subsequent parts.

First, we define what it means for a flat input to remain *stable*. We consider an input *stable* if the context in which the different input parts are interpreted remains the same. For example, assume that a sequence of bytes of an input to our 3D printer has been interpreted as GCode instructions. Now, when the fuzzer mutates the input, the same bytes should still be interpreted as GCode (if this is the case, the input is *stable*). If, however, some of these GCode bytes are now interpreted differently (e. g., as a GPIO output of a magnetic sensor instead of GCode), the semantic meaning of the input changes: The input likely no longer contains a valid GCode sequence (we say the input is *unstable*).

As a result, for a flat input to remain stable, the following properties need to hold:

1. The *amount of accesses* performed in a given context needs to remain unchanged.
2. The *order of accesses* across different contexts needs to remain unchanged.

If a given mutation changes the amount or the order of accesses performed by the firmware, the flat input becomes *unstable*. To see how simple mutations may cause a flat input to become unstable in different ways, we consider a single bit flip—a mutation that is commonly applied during fuzzing—in two different locations of the example input. These two mutations are visualized in Figure 3.


(a) Flat input representation (unmodified)

(b) Modified: Change in access amount

(c) Modified: Change in access order (via timing)

Figure 3: Flat input representation of the hardware accesses shown in Figure 2. Values are colored corresponding to the execution context in which they were initially read. The colored background shows the execution context in which they are read. A mismatch in colors indicates an unstable input. For the unmodified input (a) the colors are identical. The effect of a single mutation is shown in (b) and (c), with the modified value marked in red. In (b) one less MMIO read is performed, therefore leaving the USART data byte to be interpreted as a temperature value (green background). In (c) additional code is executed, therefore the USART ISR immediately follows the timer ISR.

First, consider a fuzzer changing the response to the first access of the USART status register `SR` performed in `isr_usart`. Such a bit flip changes the semantic meaning of the register value from indicating that a serial data byte is available to indicating that *no* data byte is available. The effects of this change are visualized in Figure 3(b): instead of performing the additional access to the data register `DR`, `isr_usart` will return immediately and resume execution in the main loop without accessing the USART data register. Now, the main loop (green background in between 4 and 5 on the x-axis) resumes and re-interprets what has previously been a GCode character (red box) as a temperature value. Following that, the input byte previously interpreted in the main context will now be consumed within `isr_timer` (blue background), and so forth. This is an example of a change in the *amount of accesses* within an execution context leading to input instability in an avalanche effect.

A second, more subtle, source of input instability is a change in timing behavior. Changes in timing behavior affect the *order of accesses* between execution contexts due to the interleaved nature of peripheral accesses. Consider a bit flip mutation changing the peripheral response of the printhead's height sensor register (`IDR` in `isr_timer`). Instead of indicating that the sensor has not yet triggered, the height is signaled to be reached a single tick earlier. The effects of this change are visualized in Figure 3(c). It indicates an endstop state condition that has previously not been indicated at that point in time. As a consequence, `isr_timer` now takes additional time for executing further handling logic. During the extra time it takes `isr_timer` to execute, the next USART interrupt occurs, leading to `isr_usart` being executed directly after `isr_timer`, without allowing `main` to consume the value 22.

As a result, `isr_usart` consumes the value 22 in its own context (red background), leading to a similar avalanche effect and input instability as in the previous example.

In these examples, we have seen different types of avalanche effects that leave a flat input unstable. The underlying assumption made by general-purpose fuzzers (which does not generally hold for firmware) is the *spatial locality of logically-connected inputs*. In other words, general-purpose fuzzers assume that logically-connected parts of an input are located close to each other in the input. If this is the case, general-purpose mutations are effective. For example, if a size field within a network packet is encoded in four consecutive bytes within the input, the typical arithmetic mutation, which casts four bytes into an integer and subsequently decrements it to provoke an integer underflow, works well. If, on the other hand, no such spatial locality exists (i. e., the bytes of the size field are spread across the input file), this mutation becomes futile. The same concept applies to the avalanche effects of input instability; the more logically-connected parts of the input are spread across the mutated input file, the more significant are the avalanche effects of input instability.

3.3 Shortcomings of Flat Inputs

As a consequence of processing inputs in a flat, sequential manner, we identify three main challenges that block an effective fuzzer integration:

Loss of Progress. Due to the instability of flat inputs, fuzzers often discard meaningful semantics while mutating. As a result, it is difficult for a fuzzer to progress further, especially when firmware enforces complex requirements on the format of its inputs.

Missing Type Information. The opaque nature of the flat binary input hides information available to the emulator from the fuzzer, as the fuzzer cannot relate a specific input byte to type information. This includes the sizes of MMIO register accesses and the order in which the accesses occur. Without this information and feedback on how differently-typed inputs are used, the fuzzer is unable to perform context- and type-aware mutations.

Inapplicable Mutations. As firmware continuously interleaves its accesses to peripherals, logically-connected parts are scattered across the flat binary input. This invalidates many established types of mutations, which implicitly assume that logically-connected parts of the input are located at adjacent input bytes. Examples include the insertion of tokens from a dictionary and arithmetic operations on adjacent bytes, such as a type cast of four bytes as an integer followed by a subtraction or an addition (which have shown to uncover issues such as integer underflows/overflows).

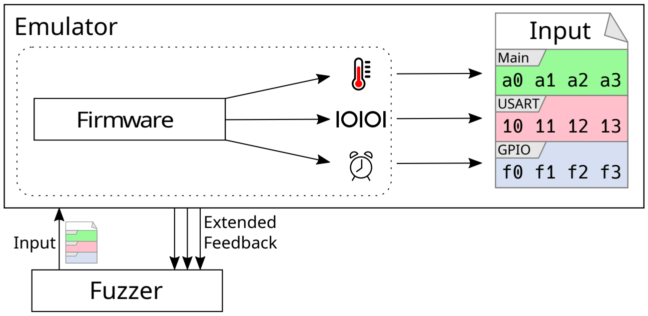


Figure 4: High-level overview of HOEDUR's design. The emulator executes the target firmware in an ISA emulator (dotted). Hardware accesses by the firmware are answered by reading from the data stream that is dedicated to the accessed device. The input file is provided by the fuzzer component. In return, the emulator provides extended feedback to the fuzzer. This feedback is collected during the execution by the emulator.

4 Design

To tackle these challenges, we present HOEDUR, the first fuzzing approach that is *aware* of the underlying firmware and capable of splitting firmware inputs into multiple strictly-typed data streams. Compared to previous work that uses general-purpose and firmware-unaware fuzzers, HOEDUR can take firmware-specific execution feedback into account and operate on a multi-stream input format.

Figure 4 shows an overview of our approach. The design of HOEDUR features two distinct parts: First, we use an emulator that provides firmware-specific feedback and serves hardware values from a multi-stream input. Second, we design a firmware-aware fuzzer. In the following, we describe these two aspects in detail and highlight the unique challenges that we need to overcome with our design.

4.1 Emulator

Our emulator is based on an instruction set architecture (ISA) emulator (dotted in Figure 4) in which we load and execute the firmware under test. Whenever the firmware accesses a hardware peripheral, the emulator determines the context of that access during runtime. We define the *access context* to consist of the triple of (i) the program counter, (ii) the address of the requested MMIO register, and (iii) the access size. Including the program counter in the access context implies that instead of grouping data streams per accessed device or MMIO register, we use the more fine-grained partitioning by unique access location (i) of each MMIO register (ii). This is based on the observation that MMIO registers may be used for different purposes in different parts of the firmware code.

An example can be found in Section 3, where the firmware accesses two types of registers from the USART peripheral; the status register `SR` consists of bit flags, while the data register `DR` provides GCode characters. Although not included in the simplified source code shown in Figure 1, the firmware may check different bit flags of the status register at different code locations (data availability, error condition bits, ...).

For each of these access contexts, we create an associated data stream that provides peripheral response values. The size of the access defines the basic (byte-level) data type of the values contained within a stream, such that all streams are strictly typed. For a more fine-grained (bit-level) type granularity, we further integrate the MMIO access models introduced in previous work into our emulator [47].

The collection of all data streams represents one *multi-stream input*. As firmware is designed to run indefinitely (note the `while(true)` loop in line 2 of Figure 1), we need to determine when to stop running the emulator for a given (multi-stream) input. We terminate the emulation run once the firmware tries to access any data stream that is exhausted, which means that all response values that the data stream provides have already been consumed.

During each run, the emulator collects additional runtime information that forms the basis for our fuzzer's *firmware-aware feedback*. First, we collect edge coverage, which we modify in a firmware-specific way by eliminating edges that result from transitions between the main execution context and interrupt handling. This avoids distractions for the fuzzer, as switches between the firmware main loop and interrupt handling no longer introduce misleading coverage (such switches can occur from anywhere in the main execution context, thus quickly filling up the coverage bitmap). Second, we keep track of firmware-specific execution metrics and define their limits, including the maximum number of raised interrupts and executed basic blocks. Inputs that lead the emulation run to exceed these limits do not necessarily cause what is considered a timeout by general-purpose fuzzers. However, such inputs still take comparatively long to execute, thus slowing down fuzzing. In addition to saving time during emulation runs, a firmware-aware fuzzer may utilize the feedback on these execution metrics to optimize its input scheduling.

4.2 Mutating Multi-Stream Inputs

We base the mutation of our multi-stream inputs on well-established mutation types. These include integer arithmetic, insertions, and input splicing. We adapt these mutation types to our multi-stream inputs in different ways:

Type-awareness. With each data stream containing typed values (e. g., 8, 16, or 32-bit integers), we mutate discrete values based on their actual type. This allows us, for example, to insert a 32-bit value into a stream that is known to require a 32-bit integer. In contrast, previous fuzzers know neither the correct location to insert into nor the correct data type to use. Blindly speculating on both the location and data type leads to the destructive avalanche effects for previous fuzzers that we study in Section 3.2.

Cross-value mutations. As we have seen in Section 3.2, it is common practice in general-purpose mutations to cast a series of bytes into a larger integer and performing an arithmetic operation, thus mutating subsequent bytes as a unit. We generalize this concept to consecutive values within data streams: We combine multiple consecutive values within a data stream into a larger one, perform an arithmetic operation, and write the result back as consecutive values.

As a firmware-specific adjustment, we combine values even for wider value types (e. g., for data streams that hold 32-bit integers). We combine such wider values by extracting a single byte from each value (e. g., the least significant byte), and then combining them as usual. Combining values larger than an 8-bit integer is desirable for firmware fuzzing for the following reason: While firmware typically stores data (such as a network packet) into its parsing buffer byte-per-byte, the firmware first needs to load each value from an MMIO register. This MMIO register may be 32 bits wide, resulting in a data stream that holds 32-bit integers.

Cross-stream mutations. For multi-stream inputs, certain mutations such as input splicing (i. e., copying data from a different input into the mutated one) are meaningful in two dimensions: stream-oriented and chronological. For example, we may wish to copy data of a specific type from one data stream to another (relating to the 3D printer example, we may want to copy the GCode from another input). Here, we can simply copy consecutive values from one stream to another. In other cases, we may prefer to copy *all* peripheral interactions within a chronological time frame (e. g., to combine a specific sequence of magnetic sensor readings and a corresponding GCode snippet). For this reason, we introduce two modes to each applicable mutation: *mono* (for a single stream) and *chrono* (for a set of chronological interactions).

Input Extension. Recall that firmware is designed to run indefinitely and, therefore, expects unlimited amounts of input. It may also discover new access contexts over time, which requires dynamically adding data streams. This is in contrast to general-purpose fuzzing targets, which typically require a finite, often single input file. To successfully fuzz test firmware, we need to balance both extremes: a very short input that barely passes the firmware boot process, and an input of infinite length. For newly discovered data streams, we provide an initial set of random values. We also add a distinct mutation with which the fuzzer instructs the emulator to provide a number of random values after the multi-stream input would normally have been exhausted. This extends the input by extending its data streams. It is a further firmware-specific addition that allows the emulation to progress and avoid sudden emulation stops in an interesting code location due to an abrupt termination of the firmware input.

4.3 Scheduling Data Streams for Mutation

Before mutating an input, fuzzers naturally need to select an input to mutate. For multi-stream inputs, we additionally need to decide which data stream *within* a multi-stream input we would like to mutate. Picking a data stream to mutate is challenging for three reasons: (1) The number of data streams is high, as firmware typically accesses a range of peripheral registers, each from potentially multiple code locations. This leads to hundreds or thousands of *access contexts* and, thus, data streams. (2) Most data streams are undesirable to mutate. The reason for this is that most streams represent status registers that do not contribute to meaningful firmware functionality. While mutating these registers is important initially to find proper low-level hardware states, mutating status registers becomes ineffective beyond a certain threshold. (3) A static metric is insufficient to base our data stream scheduling on. For example, data streams may be large, yet may not influence a significant firmware state space (e. g., continuously-polled status registers or data registers for which a driver stub exists that reads from it but implements no logic). Data streams may also be important initially, but become insignificant later (e. g., status registers which are involved in the boot process or hardware initialization).

For these reasons, we require our data stream mutation scheduling mechanism to adapt over time. While initially mutating each data stream regularly, it needs to eventually prioritize data streams that hold meaningful data, such as the GCode instruction data previously mentioned in Section 3.

4.3.1 Stream Selection

As a baseline strategy, we draw data streams from their size-weighted distribution. This strategy is based on our empirical observation that the size of a data stream often (even if not always—as previously discussed) correlates with the underlying firmware state space that the data stream can influence.

Unfortunately, solely relying on data stream sizes to determine stream-picking probabilities bears the risk of the fuzzer being sidetracked by frequent yet meaningless peripheral register accesses. This is why we further use probability matching (which is also known as Thompson Sampling [10]) to adapt the probability of selecting a data stream for mutation with its *success rate* in discovering new coverage. More precisely, we calculate the ratio between the number of times a data stream has been mutated and the number of times a mutation has contributed to finding new coverage within the given input. For example, a data stream that is two times as likely to produce new coverage when mutated than another data stream will have its probability scaled by twice as much.

To avoid overfitting to a particular set of data streams, 80% of the data streams are sampled by the Thompson sampling strategy and 20% by uniform random selection. This ensures that the fuzzer picks all data streams from time to time regardless of their combined sizes and success rates.

4.3.2 Successor Stream Selection

We further adapt the stream selection schema to allow the fuzzer to mutate well-structured inputs effectively: With a 50% percent chance, we continue to mutate the data stream that was previously selected. This way, we introduce a degree of clustering whenever a data stream is mutated. This is driven by the insight that firmware processes multiple individually formatted input types (represented by data streams) at the same time. For each data stream, a single mutation alone may be insufficient to derive a new, interesting input within the format that it (implicitly) encodes. For example, a 3D printer accepts GCode-formatted serial data to determine its printing activities. As GCode itself is a well-structured format, a given piece of GCode may require multiple mutations to be transformed into another meaningful GCode instruction.

4.4 Comparison to General-Purpose Fuzzing

In summary, our design proposes the following improvements compared to state-of-the-art work on firmware fuzzing that integrate a general-purpose fuzzer via flat binary inputs:

1. Firmware-aware fuzzing (in the form of firmware-adapted coverage feedback and execution metrics, as well as a new input extension mutation that addresses the infinite runtime of firmware)
2. A multi-stream input format and stream mutation modes

Combining these techniques allows us to design a fuzzer that is well suited for analyzing firmware. It also allows us to incorporate advanced mutation techniques from general-purpose fuzzing into our firmware testing.

5 Implementation

Our implementation consists of two components: an emulator and our firmware-aware fuzzer HOEDUR. While the fuzzer logic is platform-independent, our prototype emulator targets ARM Cortex-M MCUs, as the platform is widely adopted and enables the comparison of our approach with state-of-the-art work. We will open-source our prototype implementation at https://github.com/fuzzware-fuzzer/hoedur.

5.1 Emulator

The emulator consists mainly of the ISA emulator and additional logic to respond to peripheral accesses given a multi-stream input. We adopted the existing system emulation of QEMU [4], a widely-used system emulator for multiple ISAs, as the core of HOEDUR's emulator component.

During our implementation, we made an effort to keep the modifications of QEMU itself as small as possible to allow for rebasing onto new QEMU versions in the future (205 lines added, 101 lines deleted). This is mainly achieved through building QEMU as a dynamically linked library and implementing the outside emulator logic (such as interpreting multi-stream inputs) separately in Rust [38].

We use QEMU with a dynamically configured machine that has its memory regions set up. For memory regions that are configured as MMIO, we add emulator callbacks to handle these peripheral accesses. We inject a callback at the start of each QEMU translation block to gather coverage information and fully control the execution. We use this callback to update the coverage bitmap and control execution timings. This includes enforcing execution limits and injecting interrupts deterministically. To avoid expensive process forks, we use snapshots of the emulator state, including its CPU and memory, to facilitate fast restores.

We sometimes need to extend an input with random data. This occurs when a newly discovered data stream is added or due to the explicit mutation described in Section 4.2. To extend an input, we set a limit on how many new random values are to be added to exhausted data streams in the current emulation run. We decrement this limit whenever a random value is generated and conclude the emulation run after the limit reaches zero.

5.2 Fuzzer

The overall fuzzer design is based on libFuzzer [37] with some aspects borrowed from AFL [58] and AFL++ [20]. Notable features that were adopted from libFuzzer contain various input mutators and the ENTROPIC input energy as introduced by Böhme et al. [5]. In total, our fuzzer is implemented in about 13,500 lines of Rust code.

We also adopt AFL's mutation stacking strategy by applying 4, 8, 16, or 32 mutations before executing each input. As described in Section 4.3, we adapt the probability of selecting a given data stream for mutation based on how successful it is in finding new coverage. This requires us to determine whether a mutation on a given data stream contributed to producing the new coverage. We post-process interesting inputs that produce new coverage to reduce noise within this metric. Before committing it to the corpus, we rerun the input multiple times while progressively pruning stacked mutations that do not contribute to the new coverage.

6 Evaluation

We evaluate the effectiveness of our prototype implementation of HOEDUR by considering the following research questions:

RQ 1 How does HOEDUR compare to state-of-the-art firmware fuzzers?

RQ 2 How effective are multi-stream inputs compared to a flat, single-stream input representation?

RQ 3 Can multi-stream-aware fuzzing take advantage of advanced mutations such as dictionaries?

RQ 4 Does HOEDUR find previously unknown vulnerabilities in practice?

To answer these questions, we design and conduct four different experiments.

6.1 Setup

We first outline our setup, including the used hardware, fuzzers, and firmware targets.

Hardware Configuration. All our experiments use the same hardware configuration, i. e., two Intel Xeon Gold 5320 CPUs @ 2.20GHz (52 physical cores in total), 256 GB of RAM, and SSD memory for storage.

Fuzzers. Besides HOEDUR itself, our evaluation uses the following fuzzers: FUZZWARE, SINGLE-STREAM-HOEDUR, HOEDUR+DICT, and SINGLE-STREAM-HOEDUR+DICT. FUZZWARE [47] represents the state-of-the-art in rehosting-based embedded firmware fuzzing, making it the best-performing choice to compare against. SINGLE-STREAM-HOEDUR is a version of HOEDUR that does not employ multi-stream inputs but instead consumes input in a traditional flat manner. Still, it is *firmware-aware*, i. e., it uses our improvements regarding coverage feedback and execution metrics/limits, and it features the input extension mutation. The only difference to HOEDUR is its inability to associate access contexts with separate data streams. HOEDUR+DICT and SINGLE-STREAM-HOEDUR+DICT are variants of the respective fuzzers equipped with a dictionary to showcase how multi-stream inputs impact such advanced mutation strategies.

Targets. We use two different sets of real-world firmware. Our first set is based on FUZZWARE [47]. The FUZZWARE target set consists of 12 samples that are based on Contiki-NG [14] and Zephyr [59]. These targets implement different types of network stacks and protocols (e. g., Bluetooth Low Energy, IEEE 802.15.4, 6LoWPAN, IPv6, and SNMP). Each target in this data set contains a new bug that FUZZWARE found prior to its release and for which a CVE has been assigned. We use this set to investigate HOEDUR's capability of finding vulnerabilities in real code compared to the state-of-the-art fuzzer FUZZWARE. Second, we use an established set of firmware targets following μEMU [60]. It includes samples from P2IM [18], HALUCINATOR [13], PRETENDER [23], and WYCINWYC [40]. Thus, this sample set covers a diverse range of applications, OS libraries, and hardware platforms. Some of the targets within the μEMU target set contain bugs that trivially allow an input to take control of the firmware program counter. An example is a stack-based buffer overflow vulnerability in `RF_Door_Lock`. As controlling the program counter allows a fuzzer to produce invalid coverage within its target, a valid coverage comparison is impeded. For this reason, we fix the affected bugs by introducing binary patches to these targets. We use the patched versions of the μEMU targets for all fuzzers in our experiments. We will open source these patches to the firmware fuzzing community alongside our

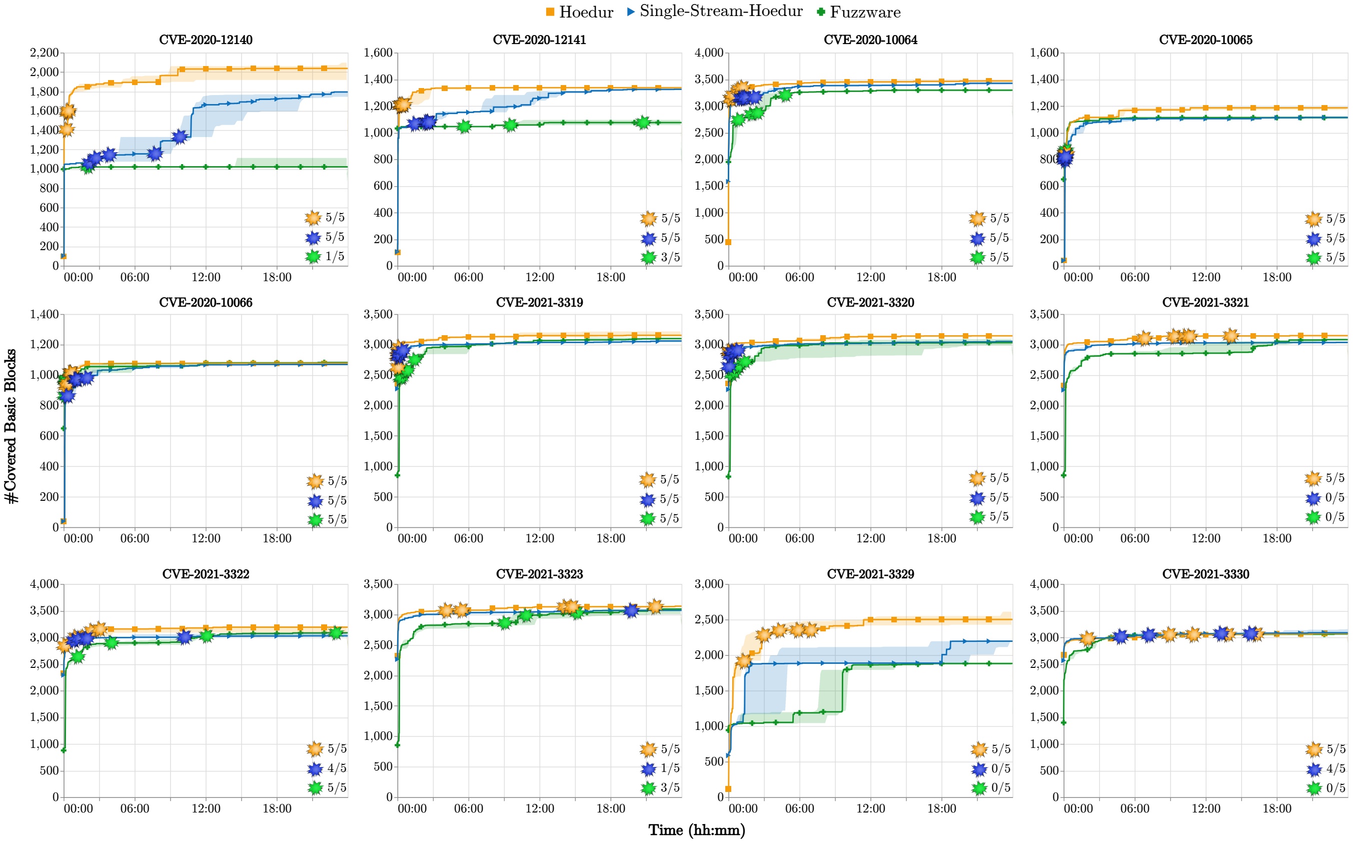


Figure 5: Coverage achieved by HOEDUR, SINGLE-STREAM-HOEDUR, and FUZZWARE on targets containing a CVE over the course of 24 hours. We plot the median and 66% interval for the five runs per fuzzer. The explosion symbol (✸, ✸, or ✸) denotes the point in time at which a run has triggered the CVE.

artifacts. To set up these targets for fuzzing in HOEDUR, we re-use equivalents to the configurations published alongside the FUZZWARE experiments. The FUZZWARE configurations for the μEMU targets contain memory mappings (i. e., ranges of RAM memory, MMIO registers, and the firmware ROM image) as well as a common interrupt raising strategy that periodically raises enabled interrupts in a round-robin fashion. For FUZZWARE's CVE target set, these configurations specify, alongside the basic configurations, a set of locations at which the emulation of an input is concluded (e. g., the `arch_system_halt` function in Zephyr) and a set of functions to be skipped (e. g., `z_impl_k_busy_wait`). To ensure a fair comparison, we wrote a tool that automatically converts FUZZWARE configurations to configurations that are compatible with HOEDUR. We do not add any further (potentially optimizing) configuration options. Providing these configurations is the only manual effort required to set up the evaluated firmware targets in HOEDUR.

6.2 Bug Finding Ability

To answer **RQ 1**, we evaluate how HOEDUR compares with FUZZWARE and SINGLE-STREAM-HOEDUR in terms of its ability to find bugs and produce code coverage. First, we run each fuzzer for 24 hours on the full set of firmware that was published alongside FUZZWARE to reproduce the CVEs that FUZZWARE found. We run each fuzzer five times as recommended by Klees et al. [32], with each run being assigned four physical CPU cores. We collect both the code coverage and CVE vulnerability discovery timings for each run. We implemented bug detection hooks for each target to determine the time at which a fuzzer triggered the CVE vulnerability. More precisely, after the fuzzing campaign has concluded, we activate the bug detection hook and rerun all crashing inputs that the respective fuzzer produced during each run. Using the detection hook, we can identify the first crashing input for which a bug trigger has been detected. We will open-source these detection hooks alongside our artifacts.

As can be seen in the plotted results shown in Figure 5, HOEDUR ties or outperforms FUZZWARE and SINGLE-STREAM-HOEDUR in terms of coverage on *all* targets. In cases where the coverage is similar, HOEDUR achieves it earlier. On average, HOEDUR finds 12% more coverage than FUZZWARE. We also observe that HOEDUR triggers the CVEs earlier than FUZZWARE. Interestingly, FUZZWARE fails to trigger the CVEs within 24 hours in nearly 40% of cases (23 not triggered out of 60). In contrast, HOEDUR triggers the CVEs within 24 hours in all cases.

Table 1: CVE discovery timings of known CVEs within 15 days. We round seconds to the closest minute and report the timings as **days:hours:minutes**. Based on the mean, we present the factor of how much faster a fuzzer is in **Fac.** (and mark the better fuzzer in bold). A fuzzer may fail to find the CVE within 15 days (see **#Hit** column); in this case, we assume it triggers the CVE at 15 days and 1 second and represent these runs as "—".

CVE-20..	Fuzzer	#Hit	Min dd:hh:mm	Max dd:hh:mm	Median dd:hh:mm	Mean dd:hh:mm	Fac.
20-12140	Fuzzware	3/5	00:02:04	—	03:08:30	07:06:27	550
	Single-Stream-Hoedur	**5/5**	00:02:12	00:09:49	00:03:53	00:05:17	17
	Hoedur	**5/5**	00:00:16	00:00:24	00:00:17	00:00:19	
20-12141	Fuzzware	4/5	00:05:40	—	00:20:46	03:14:60	236
	Single-Stream-Hoedur	**5/5**	00:01:29	00:02:40	00:02:38	00:02:22	6.4
	Hoedur	**5/5**	00:00:17	00:00:32	00:00:18	00:00:22	
20-10064	Fuzzware	**5/5**	00:00:51	00:04:50	00:02:28	00:02:59	5.0
	Single-Stream-Hoedur	**5/5**	00:01:03	00:02:14	00:01:24	00:01:29	2.5
	Hoedur	**5/5**	00:00:13	00:01:12	00:00:23	00:00:36	
20-10065	**Fuzzware**	**5/5**	00:00:06	00:00:13	00:00:07	00:00:08	0.77
	Single-Stream-Hoedur	**5/5**	00:00:08	00:00:09	00:00:08	00:00:08	0.8
	Hoedur	**5/5**	00:00:09	00:00:12	00:00:10	00:00:11	
20-10066	**Fuzzware**	**5/5**	00:00:07	00:00:13	00:00:09	00:00:10	0.32
	Single-Stream-Hoedur	**5/5**	00:00:21	00:01:57	00:01:06	00:01:07	2.2
	Hoedur	**5/5**	00:00:13	00:00:41	00:00:36	00:00:30	
21-3319	Fuzzware	**5/5**	00:00:19	00:01:27	00:00:36	00:00:43	8.6
	Single-Stream-Hoedur	**5/5**	00:00:09	00:00:28	00:00:11	00:00:14	2.8
	Hoedur	**5/5**	00:00:02	00:00:09	00:00:04	00:00:05	
21-3320	Fuzzware	**5/5**	00:00:18	00:01:25	00:00:53	00:00:49	8.0
	Single-Stream-Hoedur	**5/5**	00:00:04	00:00:48	00:00:20	00:00:21	3.5
	Hoedur	**5/5**	00:00:05	00:00:07	00:00:06	00:00:06	
21-3321	Fuzzware	2/5	01:23:15	—	—	10:11:52	25
	Single-Stream-Hoedur	**5/5**	01:00:10	08:03:37	01:17:04	02:20:38	6.7
	Hoedur	**5/5**	00:06:52	00:14:09	00:10:18	00:10:16	
21-3322	Fuzzware	**5/5**	00:01:13	00:23:00	00:04:03	00:08:53	5.3
	Single-Stream-Hoedur	**5/5**	00:00:53	01:11:34	00:01:57	00:09:59	6.0
	Hoedur	**5/5**	00:00:07	00:03:03	00:01:41	00:01:40	
21-3323	Fuzzware	**5/5**	00:09:05	04:17:58	00:15:11	01:11:59	3.0
	Single-Stream-Hoedur	**5/5**	00:19:45	02:09:34	01:12:46	01:11:53	3.0
	Hoedur	**5/5**	00:04:08	00:21:48	00:14:11	00:12:04	
21-3329	Fuzzware	3/5	01:06:38	—	06:02:13	07:20:15	44
	Single-Stream-Hoedur	**5/5**	01:00:03	11:19:50	08:04:04	06:17:37	38
	Hoedur	**5/5**	00:01:20	00:06:57	00:04:19	00:04:18	
21-3330	Fuzzware	1/5	08:23:12	—	—	13:19:02	32
	Single-Stream-Hoedur	**5/5**	00:04:50	01:05:52	00:13:18	00:14:13	1.4
	Hoedur	**5/5**	00:02:03	00:16:18	00:10:59	00:10:26	

To analyze these CVE discovery timings in more detail, we repeat the experiment but let the fuzzers run for *15 days* instead of 24 hours. The minimum, maximum, median, and mean times of triggering the CVEs are depicted in Table 1. As this table shows, Hoedur is significantly more effective and faster in triggering the CVEs than Fuzzware for all but two targets (CVE-2020-10065 and CVE-2020-10066). For these two targets, both Hoedur and Fuzzware discover the CVE within the first hour in all cases. Interestingly, Fuzzware fails to trigger the CVE for some runs even after running for 15 days. This is in stark contrast to Hoedur, for which the longest run took 25 hours (CVE-2021-3323).

While analyzing the crashes found by Hoedur, we further investigate whether Hoedur identified additional, previously unknown bugs in the Fuzzware target set, which have not been reported by previous work. To this end, we manually triaged crashes that were not caught by our CVE bug detection hooks, and we iteratively added detection hooks. Our triaging uncovered a total of 8 bugs that have not been detected by previous work and had been unfixed in the most recent versions of the underlying operating systems. We reported these bugs to their respective vendors in a coordinated way.

Table 2: Timings of the discovery (within 15 days) of additional, previously unknown bugs in the Fuzzware target set. The format is equivalent to Table 1.

CVE-20..	Fuzzer	#Hit	Min dd:hh:mm	Max dd:hh:mm	Median dd:hh:mm	Mean dd:hh:mm	Fac.
23-0397	Fuzzware	**5/5**	00:21:24	12:14:17	02:11:54	04:05:46	47
	Single-Stream-Hoedur	**5/5**	00:11:24	11:20:09	01:15:60	03:12:18	39
	Hoedur	**5/5**	00:00:59	00:02:53	00:02:51	00:02:11	
23-1422	Fuzzware	0/5	—	—	—	—	131
	Single-Stream-Hoedur	0/5	—	—	—	—	131
	Hoedur	**5/5**	00:00:33	00:05:17	00:01:57	00:02:45	
23-1423	Fuzzware	4/5	01:22:25	—	05:20:03	07:22:14	48
	Single-Stream-Hoedur	3/5	00:10:05	—	01:12:00	06:13:55	40
	Hoedur	**5/5**	00:02:27	00:06:49	00:03:00	00:03:57	
23-1901	**Fuzzware**	4/5	00:00:16	—	00:21:13	03:16:05	0.34
	Single-Stream-Hoedur	0/5	—	—	—	—	1.4
	Hoedur	2/5	01:10:22	—	—	10:17:21	
23-1902	**Fuzzware**	1/5	01:20:45	—	—	12:08:57	0.98
	Single-Stream-Hoedur	0/5	—	—	—	—	1.2
	Hoedur	1/5	03:00:38	—	—	12:14:32	
23-23609	Fuzzware	0/5	—	—	—	—	818
	Single-Stream-Hoedur	**5/5**	00:04:44	01:03:36	00:13:33	00:15:17	35
	Hoedur	**5/5**	00:00:20	00:00:34	00:00:28	00:00:26	
23-28116	Fuzzware	1/5	13:15:30	—	—	14:17:30	231
	Single-Stream-Hoedur	**5/5**	00:04:53	01:03:43	00:10:55	00:14:48	9.7
	Hoedur	**5/5**	00:00:35	00:02:60	00:01:01	00:01:32	
23-29001	Fuzzware	0/5	—	—	—	—	51
	Single-Stream-Hoedur	**5/5**	00:22:39	05:05:56	01:18:15	02:05:42	7.6
	Hoedur	**5/5**	00:02:34	00:16:44	00:05:35	00:07:04	

Table 2 shows the discovery timings of these bugs. Overall, Fuzzware found 5 of the 8 previously unknown bugs within 5 full 15-day iterations. On average, Hoedur took 166x less time to discover each bug. For 24-hour runs, Fuzzware would have found 2 out of 8 bugs within 5 iterations, while Hoedur would have found 6 out of 8.

> **RQ 1 – State of the Art:** Hoedur achieves better code coverage than the state-of-the-art fuzzer Fuzzware and—at the same time—triggers the CVE vulnerabilities significantly faster. Additionally, it found several new bugs in already fuzzed firmware.

6.3 Code Coverage: Established Data Set

To confirm whether our initial observation about Hoedur's ability to find more code coverage (**RQ 1**) scales to a larger set of targets, we design a second experiment. Here, we measure the coverage for Hoedur and Fuzzware on the dataset used by μEmu [60]. To further analyze the impact of our most significant design decision, multi-stream inputs (**RQ 2**), we again include Single-Stream-Hoedur. We run each fuzzer ten times and plot the median as well as the 66% interval in Figure 6. For brevity, we only plot the eight most complex targets (more specifically, the targets for which the fuzzers found the most coverage). Plots for the remaining targets of the dataset can be found in Figure 8 in the Appendix.

As Figure 6 shows, both Hoedur and Single-Stream-Hoedur clearly outperform Fuzzware. This is an interesting result, indicating that our fuzzer's *awareness* of the underlying firmware is beneficial. We analyzed two examples where the single-stream version Single-Stream-Hoedur clearly outperforms Fuzzware: `6LoWPAN Receiver` and

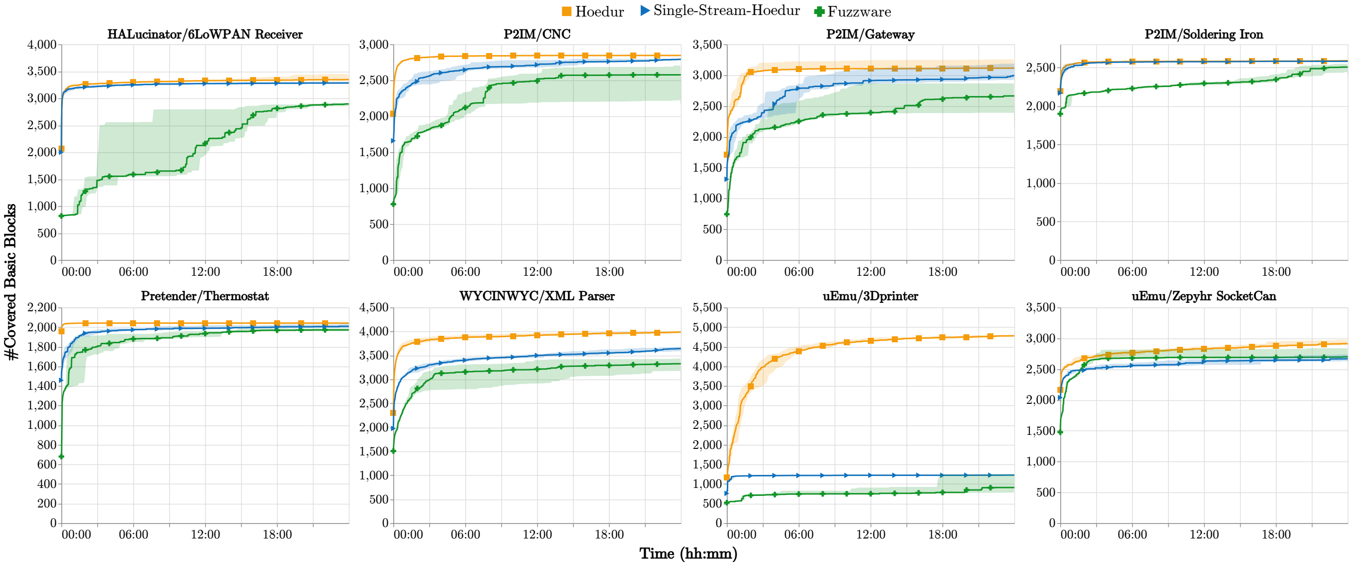


Figure 6: Coverage achieved by HOEDUR and FUZZWARE on a diverse set of targets for ten 24-hour runs. We plot the median and 66% interval.

`Soldering Iron`. We found that due to the frequent execution context switches that occur during interrupt handling, FUZZWARE misleadingly indicates a lot of new coverage, which distracts its fuzzer. As we prune these coverage edges for interrupts (see Section 4.1), this is not the case for SINGLE-STREAM-HOEDUR. Consequently, SINGLE-STREAM-HOEDUR explores the firmware more quickly and without getting distracted by superfluous coverage. While FUZZWARE catches up over time, it requires much time to do so. One notable exception, where firmware awareness does not help much is `Zephyr Socket Can`, where FUZZWARE outperforms SINGLE-STREAM-HOEDUR (but not HOEDUR) by a small margin. In this case, the target contains a common interrupt handler wrapper function, which naturally minimizes the number of distinct misleading coverage edges. HOEDUR, with its ability to assign a separate data stream to each access context, performs best overall. Especially for `3Dprinter`, it finds more than five times the coverage of FUZZWARE. On average, HOEDUR finds 32.5% more coverage (and SINGLE-STREAM-HOEDUR 9.6% more coverage) than FUZZWARE.

This validates our response to **RQ 1**: HOEDUR outperforms the state-of-the-art fuzzer FUZZWARE, both in terms of code coverage and especially in its ability to trigger bugs quickly.

In the following, we study the difference between HOEDUR and SINGLE-STREAM-HOEDUR to answer **RQ 2**. Interestingly, the multi-stream input representation is highly beneficial in some cases (`3Dprinter` and `XML Parser`), but seems to provide little benefit in other cases (`6LoWPAN Receiver` and `Soldering Iron`). Manually inspecting these targets, we found these target-specific differences to result from a concept we discussed in Section 3.2, namely, the *spatial locality of logically-connected parts of the input*. For `6LoWPAN Receiver` and `Soldering Iron`, the contained firmware logic consumes its inputs in chunks. Their respective main input functions, `trx_sram_read` and `HAL_I2C_Mem_Read`, read their input into a buffer in a loop without changing the execution context. This leads to a natural spatial locality for these two targets. Consequently, a flat single-stream approach does not suffer from the drawbacks discussed in Section 3. In contrast, `3Dprinter` represents the complete opposite: Here, the serial GCode characters are read one byte at a time, where a byte is added each time the USART interrupt occurs. This leads to the GCode characters being spread across the flat input, making it prohibitively hard for a fuzzer to mutate it effectively without the concept of multi-stream inputs.

RQ 2 – Multi-stream Inputs: Using multi-stream inputs and its differentiation of access contexts provides *robust coverage* for *all* the targets we explored. Depending on how a given firmware processes its inputs under the hood, our multi-stream input representation allows the fuzzer to vastly outperform its flat input equivalent (finding 3.86 times the coverage of SINGLE-STREAM-HOEDUR for `3Dprinter` and 20.9% more in general).

6.4 Advanced Mutations via Dictionaries

The awareness of multi-stream inputs not only benefits the fuzzer by itself, but it also unlocks its capabilities to work with advanced mutations. One compelling way of improving fuzzing performance on targets expecting a large number of specific (string) tokens is using *dictionaries*.

To verify whether multi-stream inputs allow the fuzzer to use such advanced mutation types, we run HOEDUR

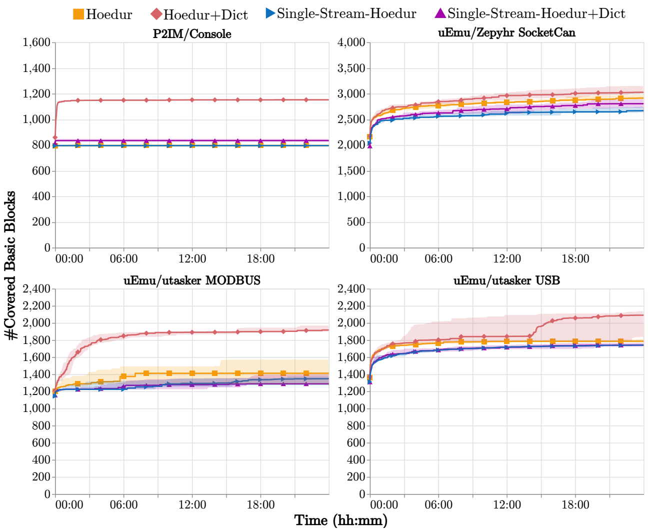


Figure 7: Coverage achieved by Hoedur+Dict and Single-Stream-Hoedur+Dict (compared to the baseline, Hoedur) on four targets performing string matching to different degrees. We performed ten runs of 24 hours each. We plot the median and 66% interval for each fuzzer.

and Single-Stream-Hoedur both without and with access to the same target-specific dictionaries (the latter variants are called Hoedur+Dict and Single-Stream-Hoedur+Dict, respectively). We automatically generated these dictionaries by extracting tokens from the respective firmware ROM image. The tokens were created based on each sequence of printable, newline, or tabulator characters with a length of 4 to 64 bytes, of which 75% represent alphanumeric or punctuation characters. Note that we have not applied these dictionaries in the previous experiments to ensure a clean comparison between the fuzzers.

We run each fuzzer ten times on four targets that make use of string matching: `Console`, `Zephyr SocketCan`, `utasker modbus`, and `utasker USB`. Figure 7 shows the median coverage and 66% intervals. Evidently, applying a dictionary increases the firmware coverage for fuzzing with a multi-stream input representation in every case (Hoedur+Dict outperforms Hoedur in every case, on average by 16.2%). At the same time, dictionary mutations provide only a much more limited advantage for the single-stream case; Single-Stream-Hoedur+Dict slightly outperforms Single-Stream-Hoedur on two targets and ties with it in the other cases. On average, Single-Stream-Hoedur+Dict finds 2.8% more coverage than without a dictionary, a much smaller difference than for Hoedur.

RQ 3 – Advanced Mutations: Our multi-stream input representation allows firmware fuzzing to effectively use advanced mutations, which have previously been of limited use in an ad-hoc flat input fuzzer integration.

6.5 Finding Unknown Vulnerabilities

In Section 6.2, we have shown that Hoedur is able to find new bugs, even previously unknown ones in the exact firmware images that have been tested in previous work. We expand on this by performing additional testing. We use Hoedur to fuzz additional targets in their most recent versions. We include different network stack implementations of four popular projects in this experiment: Zephyr OS, Contiki-NG, RIOT OS, and LoRaMac-node. Note that Fuzzware was previously used to test Zephyr and Contiki-NG.

Based on the fuzzing results of these new targets, we disclosed an additional 15 previously unknown bugs to their respective vendors, which results in an overall total of 23 newly disclosed vulnerabilities. Table 3 in the Appendix provides an overview of these vulnerabilities.

In the following, we analyze the impact of our multi-stream input representation on Hoedur's ability to discover bugs more quickly than previous work. We consider three case studies, each from a different category: 3Dprinter from μEmu (see Section 6.3), CVE-2021-3329 from the Fuzzware target set (Section 6.2), and CVE-2023-1422, which is a previously unknown bug in Zephyr found by Hoedur (see Table 2).

uEmu/3Dprinter. In this target, the 3Dprinter GCode input is read by the firmware byte by byte, while each byte is added to the input buffer during handling of a separate interrupt. This results in the GCode bytes being spread across the input file, which breaks the *spatial locality* of the GCode input bytes in a flat binary input format, making it prone to destructive avalanche effects. Hoedur, in contrast, holds these GCode input bytes in a separate data stream and is thus able to perform mutations unimpeded.

CVE-2021-3329. To trigger this vulnerability, the fuzzer must craft a series of Bluetooth HCI frames to correctly initialize the Bluetooth stack. Considering a previous input that already contains a valid part of the initialization sequence, deriving an improved input is difficult for a flat input format: A set of mutations must first be fortunate enough to introduce the necessary changes to one or more HCI frames. It *also* needs to leave the existing HCI frames intact, which is unlikely for a set of stacked mutations on a flat input. Hoedur's multi-stream input format, on the other hand, will not accidentally destroy existing HCI frames with unrelated mutations.

CVE-2023-1422. This bug is a race condition in the input/output behavior of the Zephyr Bluetooth stack. Triggering it introduces an additional layer of complexity: It requires specific HCI frames to be sent in a particular order and in quick succession. Thus, a rather large input needs to be mutated and needs to stay intact, meaning that the avalanche effect needs to be fully avoided.

RQ 4 – New Bugs: HOEDUR has found 23 previously unknown vulnerabilities in firmware, some of which has been fuzzed extensively by previous work. These findings underline the capability of HOEDUR to find security-relevant bugs.

7 Discussion

In the following, we discuss our multi-stream approach and how our prototype could be integrated into other work.

Avalanche Effect. In Section 3.2, we described the destructive avalanche effect that causes mutations of a flat input representation to discard fuzzing progress. We see this effect with virtually any firmware. Its impact, however, depends on how firmware reads its input and on the size, complexity, and depth of the structure that the input must maintain. If firmware logic is trivial (as is the case for P2IM/Heat_Press or P2IM/PLC), any input representation quickly explores all firmware behavior. However, this is different for more complex issues: Triggering CVE-2021-3321 requires multiple matching 6LoWPAN fragments and, similarly, CVE-2021-3329 requires multiple HCI setup messages to stay intact. While a fuzzer can still find more shallow bugs despite this avalanche effect, multi-stream inputs trigger bugs more quickly and reliably, even if they are harder for a fuzzer to discover.

It is worth noting that there might be situations where the instability introduced by the avalanche effect may provide some benefits. For example, it may be beneficial in certain situations to group all values of a status register into a single stream instead of creating a separate stream for each unique access to the status register. Although we have empirically found such situations to have little impact, future work may address these edge cases.

Integration of Orthogonal Approaches. While the multi-stream input format is strongly coupled with our firmware-aware fuzzer, the integration of an emulator is rather loosely coupled to allow for adopting other improvements. We added a flexible layer to the data stream handling of our design to easily integrate modeling approaches of other work. As described in Section 4, we demonstrate this by integrating the models introduced by FUZZWARE [47], which further increased HOEDUR's effectiveness. Furthermore, the firmware-aware feedback from the emulator is used optionally; the emulator implementation is only required to provide a coverage bitmap and serve our multi-stream input. It is therefore possible to replace the emulator implementation completely and adapt other work to use HOEDUR.

8 Related Work

In recent years, fuzzing has been a very active field of research, mainly started by the release and great success of the coverage-guided greybox fuzzer AFL [58]. Follow-up work has further improved it or used its concepts to create new fuzzers such as libFuzzer [37]. Improvements include mutations [1, 2, 24, 42, 44], scheduling algorithms [5–7, 9, 46, 54], and feedback [21, 27, 35, 36]. Other work has focused on increasing the input quality to overcome roadblocks by using taint tracking [12,45] or symbolic execution [22, 51, 56]. While many fuzzers target only desktop applications, there are adoptions to new fields such as kernels [26, 50, 53] and hypervisors [8, 25, 43, 48, 49]. None of these improvements target monolithic firmware.

As outlined in this work, our approach automatically subdivides the firmware input into multiple data streams. Grouping data into independent streams and exposing these streams separately to the fuzzer potentially opens up firmware fuzzing to these approaches. Previous work also operates on inputs containing different logical units [3,19,44,49,53], albeit targeting different domains. Closer to hardware, a recent approach proposed using multiple streams for USB kernel fuzzing [31]. Other than HOEDUR, this work utilizes Linux kernel abstractions and requires previous knowledge of the different input channels, which is not feasible for monolithic firmware.

Previous work in firmware fuzzing has come a long way from black-box [11, 34, 41] and hardware-in-the-loop [15, 29, 30, 33, 39, 52, 57] fuzzing. For all approaches requiring physical hardware during any stage of testing, providing introspection and effectively scaling analyses to large computation power has proven to be a major challenge.

Recent state-of-the-art work solved these problems using rehosting [17, 18, 47, 55, 60]. Existing rehosting approaches use AFL [58] as their drop-in fuzzer component: P2IM [18], μEMU [60] and FUZZWARE [47] all use AFL without changes to its fuzzing logic. Instead, they all focus on emulator improvements that are orthogonal to our firmware-aware fuzzer. In contrast, HOEDUR targets an aspect which was left for optimization by previous work, namely, making the fuzzer aware of the underlying firmware. This includes improving the interaction between the firmware emulator and the integrated fuzzer as well as the interpretation of fuzzing input as multiple data streams.

9 Conclusion and Future Work

In this work, we identify a number of challenges encountered by the ad-hoc integration of general-purpose fuzzers for testing firmware. Based on these insights, we devise novel firmware-aware fuzzing techniques that feature a multi-stream input representation. Using our multi-stream inputs, we unlock effective mutations of firmware input. We show that HOEDUR significantly outperforms the state of the art in embedded firmware fuzzing, both in terms of coverage and particularly in its ability to find more bugs more quickly.

As our multi-stream input representation allows for mutating firmware fuzzing input analogous to how general-purpose fuzzers mutate inputs for testing user-space applications, our multi-stream technique potentially opens up firmware fuzzing to a wide field of exciting improvements. Consider, for example, a UART device that provides shell access, where the MMIO register with the input data is mapped to one data stream. A specialized grammar fuzzer can provide this data stream without interference from other MMIO accesses.

Acknowledgements

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research under the grant CPSec (16KIS1899). This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

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

[5] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[8] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (Input) Space to Fuzz Virtual Devices. In *USENIX Security Symposium*, 2022.

[9] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive Mutational Fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[10] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2011.

[11] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[12] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy*, 2018.

[13] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting through Abstraction Layer Emulation. In *USENIX Security Symposium*, 2020.

[14] Contiki-NG. `https://github.com/contiki-ng/contiki-ng`, 2020. Accessed: October 11, 2022.

[15] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *USENIX Security Symposium*, 2018.

[16] Erik van der Zalm et al. Marlin Firmware. `https://marlinfw.org/`, 2022. Accessed: October 11, 2022.

[17] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[18] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *USENIX Security Symposium*, 2020.

[19] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[21] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[22] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based Whitebox Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[23] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[24] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[25] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017.

[26] Jesse Hertz and Tim Newsham. Project Triforce: Run AFL on Everything! *NCC Group, Tech. Rep.*, 2016.

[27] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[28] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *USENIX Security Symposium*, 2021.

[29] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded Security Testing with Peripheral Device Caching and Runtime Program State Approximation. In *Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[30] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: Peripheral Proxying Supported Embedded Code Testing. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[31] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin R. B. Butler, Antonio Bianchi, and Dave Jing Tian. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In *IEEE Symposium on Security and Privacy*, 2022.

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

[33] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[34] Karl Koscher, Stefan Savage, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Alexei Czeskis, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental Security Analysis of a Modern Automobile. In *IEEE Symposium on Security and Privacy*, 2010.

[35] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/. Accessed: October 11, 2022.

[36] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017.

[37] LLVM. libFuzzer – a Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html. Accessed: October 11, 2022.

[38] Nicholas D Matsakis and Felix S Klock II. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[39] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research (BAR)*, 2018.

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

[41] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *USENIX Security Symposium*, 2011.

[42] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Validity Fuzzing and Parametric Generators for Effective Random Testing. In *International Conference on Software Engineering (ICSE)*, 2019.

[43] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[44] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 2019.

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

[46] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium*, 2014.

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

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

[49] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

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

[51] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[52] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *USENIX Security Symposium*, 2018.

[53] Dimitri Vyokov. Syzkaller. https://github.com/google/syzkaller. Accessed: October 11, 2022.

[54] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling Black-box Mutational Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[55] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in Firmware Re-Hosting, Emulation, and Analysis. *ACM Computing Surveys (CSUR)*, 2021.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

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

[58] Michal Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017. Accessed: October 11, 2022.

[59] Zephyr Project. https://www.zephyrproject.org/, 2022. Accessed: October 11, 2022.

[60] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *USENIX Security Symposium*, 2021.

A Appendix

We list the bugs found by HOEDUR that were unknown to date in Table 3. Additionally, we plot the coverage for HOEDUR, SINGLE-STREAM-HOEDUR, and FUZZWARE for the remaining targets of the μEMU [60] dataset in Figure 8. Table 4 contains the size information of the target firmware contained within our target set.

Table 3: Overview of previously unknown bugs found by HOEDUR. During fuzzing the firmware images that were previously fuzzed by FUZZWARE, HOEDUR also found bugs that were unknown to date. All bugs have been responsible disclosed, however, many remain unfixed and thus were redacted for submission. More information will be included once the vulnerabilities have been fixed by the vendors.

CVE-20..	Target	Version	Description
22-41873	Contiki-NG	4.8	l2cap_channels OOB in get_channel_for_cid
22-41972	Contiki-NG	4.8	input_l2cap_credit missing NULL pointer check
23-31129	Contiki-NG	4.8	Missing Null Pointer Check in IPv6 Neighbor Discovery
23-29001	Contiki-NG	4.8	Infinite Recursion for IPv6 Routing Header
23-23609	Contiki-NG	4.8	Improper size validation of L2CAP frames
23-28116	Contiki-NG	4.8	mac_max_payload invalidating 6lo output bounds checks
23-24819	RIOT	2022.07	Buffer Overflow during IPHC receive
23-24820	RIOT	2022.07	Integer Underflow during IPHC receive
23-24821	RIOT	2022.07	Integer Underflow during defragmentation
23-24822	RIOT	2022.07	Null Pointer dereference during IPHC encoding
23-24823	RIOT	2022.07	Packet Type Confusion during IPHC send
23-24818	RIOT	2022.07	Null Pointer dereference during fragment forwarding
23-24825	RIOT	2022.07	Null pointer dereference in gnrc_pktbuf_mark
23-24826	RIOT	2022.07	Usage of Uninitialized Timer during forwarding of Fragments with SFR
23-24817	RIOT	2022.07	Out of Bounds write in routing with SRH
22-39274	LoRaMac-node	4.6	Buffer Overflow in ProcessRadioRxDone
22-3806	Zephyr	3.1	Double Free in bt_spi_send in Error case
23-0359	Zephyr	3.2	Missing Null pointer check in IPv6 processing
23-0397	Zephyr	3.2	Missing data checks can lead to invalid initialization
23-1422	Zephyr	3.2	sent_cmd Shared Reference Race Condition
23-1423	Zephyr	3.2	HCI Priority Event Handling Misses Allocation Error Handling
23-1902	Zephyr	3.2	HCI Connection Creation Dangling State Reference Re-use
23-1901	Zephyr	3.2	HCI send_sync Dangling Semaphore Reference Re-use

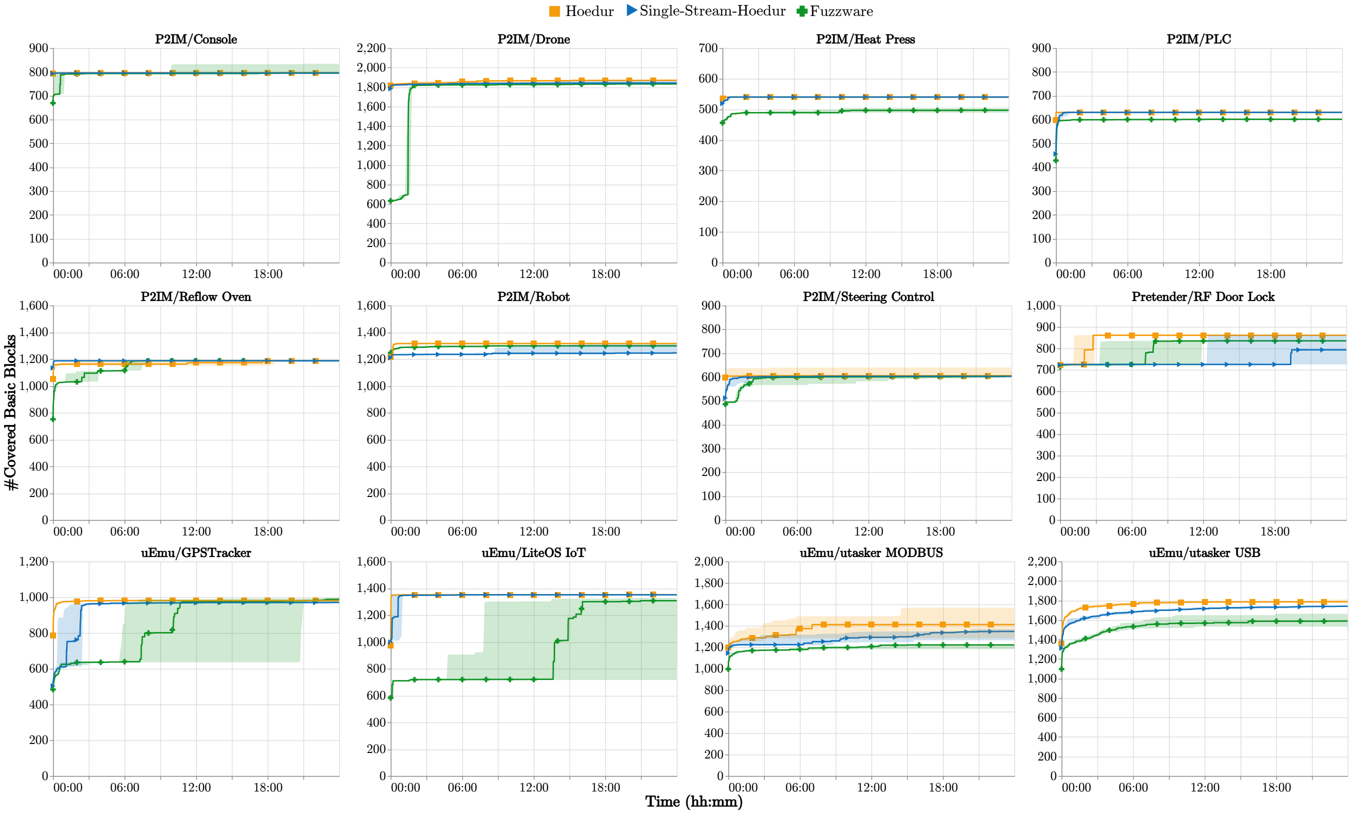


Figure 8: Coverage achieved by HOEDUR and FUZZWARE on a diverse set of targets for ten 24-hour runs. We plot the median and 66% interval. Many of these targets have a small amount of coverage, most of which is detected almost instantly by all fuzzers.

Table 4: Experiment Dataset Details. This table shows the size of the tested binary code (**Size**) and the number of basic blocks contained within each target (**#BBs**).

Target	**Size**	**#BBs**
Fuzzware/contiki-ng/CVE-2020-12140	504kB	4,002
Fuzzware/contiki-ng/CVE-2020-12141	504kB	3,080
Fuzzware/zephyr-os/CVE-2020-10064	84kB	7,316
Fuzzware/zephyr-os/CVE-2020-10065	51kB	4,949
Fuzzware/zephyr-os/CVE-2020-10066	51kB	4,951
Fuzzware/zephyr-os/CVE-2021-3319	79kB	7,009
Fuzzware/zephyr-os/CVE-2021-3320	79kB	7,007
Fuzzware/zephyr-os/CVE-2021-3321	79kB	7,003
Fuzzware/zephyr-os/CVE-2021-3322	79kB	7,006
Fuzzware/zephyr-os/CVE-2021-3323	79kB	7,006
Fuzzware/zephyr-os/CVE-2021-3329	50kB	4,972
Fuzzware/zephyr-os/CVE-2021-3330	79kB	6,911
HALucinator/6LoWPAN_Receiver	69kB	6,977
P2IM/CNC	49kB	3,614
P2IM/Console	29kB	2,251
P2IM/Drone	29kB	2,728
P2IM/Gateway	43kB	4,921
P2IM/Heat_Press	24kB	1,837
P2IM/PLC	24kB	2,303
P2IM/Reflow_Oven	30kB	2,947
P2IM/Robot	41kB	3,034
P2IM/Soldering_Iron	65kB	3,656
P2IM/Steering_Control	24kB	1,835
Pretender/RF_Door_Lock	38kB	3,320
Pretender/Thermostat	54kB	4,673
WYCINWYC/XML_Parser	92kB	9,376
uEmu/3Dprinter	88kB	8,045
uEmu/GPSTracker	46kB	4,194
uEmu/LiteOS_IoT	29kB	2,423
uEmu/Zepyhr_SocketCan	78kB	5,943
uEmu/utasker_MODBUS	40kB	3,780
uEmu/utasker_USB	36kB	3,491