MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware

Michael Chesser
The University of Adelaide
Data61 CSIRO, Cyber Security Cooperative Research Centre

Surya Nepal
Data61 CSIRO, Cyber Security Cooperative Research Centre

Damith C. Ranasinghe
School of Computer Science
The University of Adelaide

Abstract

Rapid embedded device proliferation is creating new targets and opportunities for adversaries. However, the complex interactions between firmware and hardware pose challenges to applying automated testing, such as fuzzing. State-of-the-art methods re-host firmware in emulators and facilitate complex interactions with hardware by provisioning for inputs from a diversity of methods (such as interrupts) from a plethora of devices (such as modems). We recognize a significant disconnect between how a fuzzer generates inputs (as a monolithic file) and how the inputs are consumed during re-hosted execution (as a stream, in slices, per peripheral). We demonstrate the disconnect to significantly impact a fuzzer's effectiveness at discovering inputs that explore deeper code and bugs.

We *rethink* the input generation process for fuzzing monolithic firmware and propose a new approach—*multi-stream input generation and representation*; inputs are now a collection of independent streams, one for each peripheral. We demonstrate the versatility and effectiveness of our approach by implementing: i) stream specific mutation strategies; ii) efficient methods for generating useful values for peripherals; iii) enhancing the use of information learned during fuzzing; and iv) improving a fuzzer's ability to handle roadblocks. We design and build a new fuzzer, MultiFuzz, for testing monolithic firmware and evaluate our approach on synthetic and real-world targets. MultiFuzz passes all 66 unit tests from a benchmark consisting of 46 synthetic binaries targeting a diverse set of microcontrollers. On an evaluation with 23 real-world firmware targets, MultiFuzz *outperforms* the state-of-the-art fuzzers Fuzzware and Ember-IO. MultiFuzz reaches significantly more code on 14 out of the 23 firmware targets and similar coverage on the remainder. Further, MultiFuzz discovered 18 *new* bugs on real-world targets, many thoroughly tested by previous fuzzers.

1 Introduction

With the availability of lower cost, lower power and smaller computing devices, we are witnessing a proliferation in *smart* devices. Now, microcontrollers running firmware are becoming integral components of safety and security critical systems. Firmware on embedded systems integrates and interacts with a diverse set of peripheral devices, such as modems or serial ports, while managing communications with the user, often without any supervisory control from an operating system. *Importantly*, the lack of supervisory control reduces the ability to detect faults and abort in a controlled manner [38] with potential safety and security implications. So, a scalable and automated method to identify software bugs and vulnerabilities before public release is a research and societal imperative.

Fuzzing is a de-facto industry standard for software testing and can play a crucial role in developing secure connected devices through scalable and automated testing of firmware. To test software, fuzzers automatically generate and execute inputs to uncover unusual program behavior. However, the unique characteristics of embedded devices and their firmware present challenges for adopting fuzzing tools [38]. To overcome these challenges, it is becoming increasingly common to execute firmware in an emulated environment in a process known as re-hosting [15].

When firmware runs on top of a standard operating system, re-hosting is possible by emulating operating system abstractions [6,30,48,52,53]. Unfortunately, this approach cannot be applied to fuzz monolithic firmware, which directly interacts with a large number of peripherals through memory-mapped Input/Output (MMIO) as shown in Figure 1. Fuzzing this type of firmware effectively requires supplying values for each peripheral register whenever accessed [1]

Handling MMIO Accesses. One solution for dealing with MMIO accesses is to perform hardware-in-the-loop testing, where each access is forwarded to physical hardware [11,12,37,38]. However, interacting with physical hardware can be complex and slow, introduces restrictions on how inputs can be manipulated, and inhibits the fuzzer's scalability. Alternatively, high-level emulation approaches enable hardware-

[1]Our work focuses strictly on monolithic firmware, for simplicity, the term 'firmware' refers to 'monolithic firmware' in the remainder of the paper.

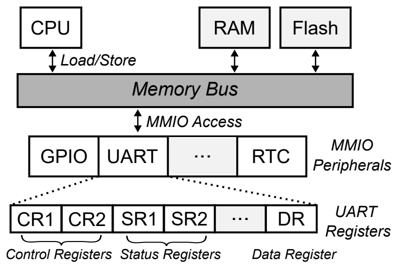


Figure 1: An overview of how the CPU of a microcontroller (MCU) interacts with memory-mapped (MMIO) peripherals. Monolithic firmware fuzzing requires supplying values for each of the peripheral registers whenever they are accessed.

free firmware re-hosting by abstracting away the hardware. These approaches avoid MMIO accesses either by emulating hardware-abstraction layers (HALs) [10, 47] or by building target-specific fuzzing harnesses [34, 44]. However, these approaches require significant engineering effort and can miss bugs in low-level code that is not executed because of the high-level emulation.

Instead, recent approaches [13, 16, 42, 54] have shown it is possible to fully automate firmware re-hosting by using a fuzzer to generate inputs for MMIO accesses to unknown peripherals. These approaches exploit the fact that *approximate* emulation of peripheral values is sufficient for the execution of most firmware. In fact, unusual values can be desirable for fuzzing, as they allow error handling code to be more thoroughly tested, which is often a source of bugs [28]. To improve the effectiveness of fuzzers based on this idea, prior work has investigated several approaches, including building peripheral models by observing MMIO access patterns [16, 35], utilizing a symbolic solver to guide execution [5, 29, 54], performing local symbolic execution to obtain precise access models that focus mutations only on the relevant bits of the input [42], and exploiting commonalities in hardware behavior to make better use of input data [13].

Input Generation and Representation. Existing re-hosting approaches use off-the-shelf fuzzing frameworks such as AFL [51] and AFL++ [18] for input generation. However, these frameworks are primarily designed to test programs that take files as inputs. The input representation used by a fuzzer can significantly impact on its ability to generate effective inputs [1, 19, 33, 45, 46]. We observe this to be the case for firmware fuzzing.

A file-based input representation is ineffective for fuzzing firmware because there is a significant disconnect between the way the input is generated and how the input is consumed during execution. Values for peripheral accesses are derived by consuming bytes from the input file. However, firmware usually interacts with multiple peripherals in an order that depends on the specific execution path. This results in data consumed by each peripheral being scattered, in an unknown manner, throughout the input file.

Consequently, mutation strategies and instrumentation techniques that assume values are accessed in contiguous blocks are less effective as even a tiny change in a sequential file input can lead to significant changes in the execution path and the subsequent mapping of bytes to peripherals. Our key insight is to recognize the inputs behave more like *streams* than files and current file-like representation limits the fuzzer's ability to take advantage of stream or peripheral specific mutations and prevents the fuzzer from effectively employing techniques that rely on inferring relationships between inputs and the state of the program during execution to solve fuzzing roadblocks.

Our Approach. Driven by these observations, we propose a multi-stream input representation for firmware fuzzing. This representation splits the input into multiple independent streams, one for each MMIO peripheral register. This is a more *natural* input representation, as it more closely reflects how firmware consumes data during execution. A multi-stream representation makes mutation operations that rely on manipulating contiguous bytes in the input, such as splicing and input-to-state replacement strategies, more effective. Additionally, by detecting when streams run out of data and employing one of three stream-specific length extension strategies, we avoid cases where the firmware gets stuck waiting for data. Further, by ensuring that streams are mutated independently, the fuzzer no longer spends a disproportionate amount of effort mutating values for MMIO peripherals that are frequently read but are uninteresting.

We implemented our approach in a new fuzzing framework, MULTIFUZZ, which we evaluate with 66 unit-tests developed by P^{2}IM [16], the 20 firmware binaries used for benchmarking in existing state-of-the-art firmware fuzzers, and 3 new real-world firmware targets. Compared to existing fuzzers, Fuzzware and Ember-IO, MULTIFUZZ achieves higher code-coverage, and uncovers 18 previously *undiscovered* bugs.

Our Contributions. In summary, we make the following contributions:

- We share limitations associated with using existing input representation methods for fuzzing firmware.
- We re-think input generation and propose a new approach—*multi-stream input generation and representation*. We design strategies for generating and mutating multi-stream inputs to solve fuzzing roadblocks and reach deeper code faster.
- We design and implement a new fuzzer, MULTIFUZZ, specifically tailored to improving firmware fuzzing by taking advantage of a multi-stream input representation. We open-source[2] our fuzzer to facilitate further research in firmware fuzzing.
- We demonstrate MULTIFUZZ outperforms existing state-

[2]Repo: https://github.com/MultiFuzz/MultiFuzz

of-the-art firmware fuzzers, Fuzzware and Ember-IO, in code coverage and bug discovery with real-world targets.

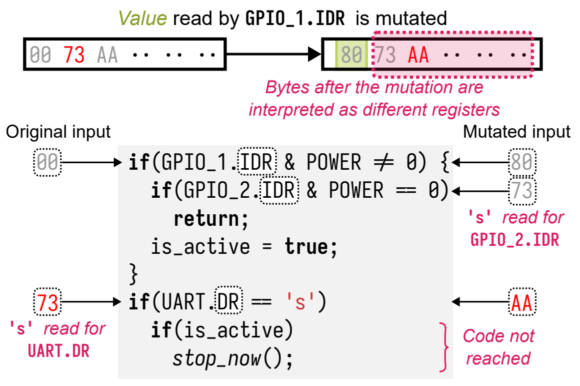
Responsible Disclosure. We disclosed the bugs found in our work in accordance with the security policies listed for the associated projects (see Appendix A.3).

2 Input Representation for MMIO Accesses

Prior to delving into the technical details of our approach, we re-visit the implications of using a file-based input representation for fuzzing re-hosted firmware to understand the limitations with existing approaches.

A Perspective on Mutational File Fuzzing for Firmware. The fuzzing frontends, AFL [51] and AFL++ [18], utilized by existing state-of-the-art firmware fuzzers [13, 16, 42, 54] are designed to test applications that take a single file as input. Therefore, to test firmware, individual MMIO accesses must be mapped back to the generated file at execution time. To do so, firmware fuzzers keep track of the position within the input to read the next value from, irrespective of the peripheral being accessed by the firmware. If there is no data left, then execution stops. Since the fuzzing frontend is unaware of this mapping, it introduces issues when mutating inputs that reduce the fuzzer's effectiveness. In this section, we describe three critical issues associated with using a file-based input representation for firmware fuzzing: Input stability, data scattering and ineffective input extensions.

Input stability. A small mutation to a file-based input can lead to highly non-local changes throughout input due to the manner in which the firmware receives input data from the fuzzer. Consider the example in Figure 2, when the value


Value read by **GPIO_1.IDR** is mutated
00 73 AA ·· ·· ··
80 73 AA ·· ·· ··
Bytes after the mutation are interpreted as different registers
Original input
Mutated input
00
80
73
's' *read for* GPIO_2.IDR
73
's' *read for* UART.DR
AA
Code not reached

```
if(GPIO_1.IDR & POWER ≠ 0) {
  if(GPIO_2.IDR & POWER == 0)
    return;
  is_active = true;
}
if(UART.DR == 's')
  if(is_active)
    stop_now();
```

Figure 2: An example of input stability issues manifesting with file-based input representation. The byte mutation in the *mutated input* changes the value read by `GPIO_1.IDR` (`0x00` to `0x80`) and causes the execution path to change. Consequently, the value originally used for the `UART.DR` (`0x73` or `s`), is now consumed by an access from a different peripheral.

(`0x00`) read from the first GPIO peripheral (`GPIO_1.IDR`) undergoes mutation, it results in the input taking a different path through the program. Consequently, the value originally read for `UART` data register (`0x73` representing an ASCII 's' character) is instead consumed when a different peripheral is accessed (the second GPIO peripheral `GPIO_2.IDR`). This leads to the firmware reading a different value (`0xAA`) for the data register, causing a later comparison that expects the value to be 's', to suddenly be false. In fact, this not only impacts the `UART` data register but potentially changes the values read for *all* future MMIO accesses. These input stability issues make it difficult for a fuzzer to make a small mutation to one part of the input without inadvertently breaking seemingly unrelated other parts of the input. Notably, the presence of interrupts further exacerbates this problem. Interrupts typically access multiple peripherals, and the scheduling of these interrupts depends on the execution path, increasing the likelihood of a significant divergence from a small change.

Data Scattering. Firmware intersperses accesses to different peripherals during execution. Using a file-based input can cause data accessed by a particular peripheral to be scattered across a large portion of the input. This can significantly hinder mutation strategies that rely on data being contiguous in the input. For example, a value is inserted from a dictionary, different bytes of the inserted value may be read by *multiple* peripherals. This also poses challenges for mutation strategies that rely on identifying associations between input bytes and values observed during executions, such as Redqueen [2]. Additionally, mutations that involve copying sub-slices of inputs, such as the *splice* operation in AFL, are also likely to be less effective, as the copied data can be consumed by different peripherals.

Ineffective input extensions. Firmware often executes continuously in a loop, waiting for data or events to occur before performing other actions. Consequently, during fuzzing, it is imperative for the fuzzer to provide *adequate* and *valid* data to the multiple registers associated with each MMIO peripheral. Existing firmware fuzzers, relying on file-based mutation strategies, do not attempt to discover effective input extensions that provide sufficient data for peripherals. Instead, these fuzzers rely *entirely* on mutation operations that only indirectly increase the size of inputs [7]. As a result, existing fuzzers experience roadblocks when firmware requires large amounts of data to be read from peripherals. Consider the function shown in Listing 1, which is used to print startup messages to a serial port. This function writes characters to the serial device one byte at a time, first checking whether the device is ready by reading a 32-bit value from the status register, then writing the byte to the data register. Some firmware may try to print large messages (e.g., exceeding 100 characters). Therefore, to make progress, the fuzzer must also extend inputs by a large number of bytes, *all* of which must indicate that the peripheral is ready to receive. In practice even larger extensions are required due to the presence of interrupts and other MMIO accesses that can occur while the print function is executing. Further, since existing input generation strategies cannot differentiate between MMIO peripherals, they are unable to inform future input extensions based on prior accesses.

```c
void serialprintPGM(char* buf) {
    while (*buf != NULL) {
        while ((UART.SR & READY) != READY);
        UART.DR = *buf;
        ++buf;
    }
}
```

Listing 1: A simplified version of the function used to printing startup messages to the console in the *3D Printer* firmware.

3 MULTIFUZZ Design

As discussed in the previous section, executing firmware relies on managing inputs for multiple MMIO peripherals. However, the normal approach to input generation based on mutating file-like inputs leads to fuzzing roadblocks and ineffective mutations, inhibiting progress. Generating and providing *effective* values for MMIO accesses is critical to building an efficient and effective fuzzer. In the following section, we introduce the design of MULTIFUZZ, a generic firmware fuzzer that removes the disconnect between the generation-time and execution-time usage of inputs to enable more effective exploration of firmware binaries.

Instead of relying on a normal file-based input representation, we propose representing inputs as a collection of *independent* streams, with each stream corresponding to an MMIO peripheral register. To make effective use of the proposed representation, we design strategies for generating and mutating multi-stream inputs. These strategies address the challenges presented in Section 2, allowing the fuzzer to reach deeper code faster. In particular:

- We develop a strategy for initializing and extending each stream within the input driven by feedback from stream exhaustion (see Section 3.1).
- We develop a series of stream-specific mutation operations: stream-to-stream splicing, mutations using a dynamic dictionary, and colorization and input-to-state replacement by adopting Redqueen for multiple-streams (see Section 3.2 and Section 3.3).
- To improve *efficiency*, we develop a strategy to prevent a disproportionate amount of mutation effort from being spent on peripheral registers with a large number of accesses and automatically reduce the effort spent mutating streams with no influence on program control flow (see Section 3.3.1).

A high-level overview of MULTIFUZZ is shown in Figure 3a. With this approach, the fuzzer can mutate and extend data for individual MMIO peripherals independently by employing a set of multi-stream specific strategies. This intuitive approach solves the problems posed by input stability, peripheral data scattering, and the ineffectiveness of input extensions in normal mutational file fuzzing, as discussed in Section 2.

3.1 Effective Input Extension

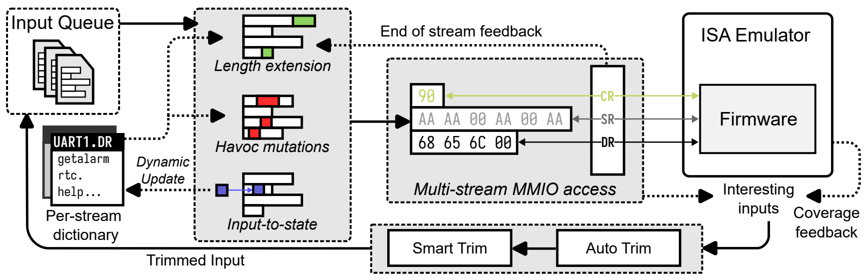
Determining the appropriate sizes for fuzzing inputs is challenging, even in normal mutational file fuzzing [2,7,20]. Inputs that are too short may fail early validation checks, while overly long inputs slow down execution and reduce the likelihood of byte-orientated mutations affecting critical bytes. One solution is to start with a *good* initial input corpus [26]. However, accomplishing this is extremely difficult for firmware since the fuzzer needs to manage MMIO accesses to a variable number of unknown peripherals. In fact, obtaining even a *single* effective starting input is challenging [29, 42, 54]. Consequently, firmware fuzzers typically start with a simple, generic input seed and heavily rely on mutations to reach deeper into the code.

As we discussed in Section 2, the file-based input representation used in existing approaches inhibits the fuzzer's ability to extend inputs effectively, reducing fuzzing performance. However, a multi-stream representation introduces new challenges, as the fuzzer must now initialize multiple streams. To address the complexity in initializing multi-stream inputs, we employ two techniques we: i) extend inputs using stream-specific length extension strategies; and ii) then utilize an improved trimming strategy to eliminate uninteresting bytes from each stream. We discuss these methods in detail, next.

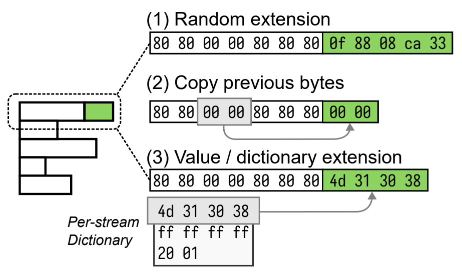
3.1.1 Multi-stream Length Extension

The core of MULTIFUZZ's approach to generating multi-stream inputs is an efficient length extension strategy. Starting with an initial value for every stream would be impractical, as it is common for a large, mostly unused, region of memory to be reserved for MMIO[3]. Instead, our approach is to start with an empty input and develop techniques for dynamically initializing streams. MULTIFUZZ first executes the firmware until it accesses a MMIO peripheral for the first time, then captures a snapshot of the emulator's state at that point. Using this snapshot, the fuzzer rapidly tests new inputs by extending the stream that the firmware failed to read, creating a new empty stream if one does not already exist. A similar approach is used to extend non-empty seed inputs, except the snapshot is taken when all the data for a stream has been consumed. When extending a stream, one of, or a combination of, the following three strategies is selected: (1) random extension, (2) copying previous bytes within the input, and

[3]For example, ARM Cortex-M reserves a 0.5 GB region for peripherals.



(a) An overview of the core components of MultiFuzz.



(b) An overview of the three length extension strategies utilized by MultiFuzz.

(3) utilizing interesting values and values from a dynamic dictionary. Figure 3b contains a visual example of each of these strategies.

Random extension. The simplest length extension strategy used by MultiFuzz is to generate random bytes and append them to the end of the target stream. This strategy is effective at extending streams for MMIO peripherals that the firmware does not expect to return exact values. As an example, firmware typically expects to read data from an Analog-to-Digital Converter (ADC) peripheral that lies within a range of outputs, so generating any value within the expected range allows the fuzzer to make progress.

Copy previous bytes. We propose extending streams by copying bytes from earlier slices within the same stream. As shown in Figure 3b (2), the fuzzer first selects a slice of bytes (`[00 00]`) from a previous location in the stream and then appends them to the end of the stream. This strategy leverages the observation [13] that peripheral registers are often interacted with in similar ways at different points during execution. Values that enable progress to be made for earlier reads are often useful to use for subsequent reads. For example, in the code presented in Listing 2, the firmware waits for the USART peripheral to finish synchronization, monitoring the SYNCBUSY register until it reads zero. To make progress, every time the firmware calls the `usart_is_syncing` function, the fuzzer must generate a value of '0' for the stream. By using a length extension strategy that duplicates previous values within the stream, subsequent calls to this function are solved more easily, improving fuzzing performance.

```c
void usart_is_syncing() {
  while (USART.SYNCBUSY != 0) {
    // Wait until synchronization is complete.
  }
}
```

Listing 2: Example of a function that waits until the value read from USART sync register is non-zero prior to continuing.

Value and dictionary extension. Streams can also be extended using a predefined set of *interesting* values or values collected into a stream-specific dictionary that is *dynamically* updated as part of the input-to-state stage (Section 3.2). For interesting values, we use a reduced set of values used by existing fuzzers [18, 19, 51]. We observed that values with a bit pattern of all ones or all zeros (i.e., the bytes `0xff` and `0x00`) are effective. This effectiveness arises from how firmware interacts with certain MMIO registers. For example, it is common for actions the firmware takes to depend on whether a specific bit within a status register is set or cleared. By extending the input with values consisting of all zeros or all ones, both possibilities can be tested without knowing the exact bit checked. In the code shown in Listing 3, the firmware only reads data from the UART peripheral after checking whether the receive ready bit is set in the status register (`UART.SR`). By extending the input with a value consisting of all ones, the fuzzer successfully progresses through this check, as the necessary bit is set. These values are also good starting points for additional mutation strategies such as bit-flipping, which are incorporated within the Havoc stage (see Section 3.3).

```c
void UARTClass::IrqHandler(void) {
  // Check if new data is available.
  uint32_t status = UART.SR;
  if ((status & RXRDY) == RXRDY) {
    // Read new data.
    store_char(UART.RHR);
  }
 // ...
}
```

Listing 3: Firmware code snippet checking a ready bit in a status register (`UART.SR`) prior to accessing the data register.

Inspired by the effectiveness of mutation stacking in AFL [20], MultiFuzz may choose to apply multiple extension strategies to a single stream. Each time MultiFuzz attempts to extend an input, it chooses a random number of extensions to apply and chooses a random strategy for each extension. Like mutation stacking in AFL, if the fuzzer fails to make any progress after many attempts, we allow the amount of stacking to increase.

After applying one or more extension strategies and executing the modified input, the fuzzer then evaluates the code-coverage for the input. Whenever an extension causes the input to reach new code, we save the mutated input for future mutations; otherwise, we revert the extension and make a new attempt from the saved snapshot.

In the code shown in Listing 3, for a new data byte to be read from the UART peripheral, *both* the `UART.SR` stream and the `UART.RHR` stream must be extended. However, if the fuzzer has found other inputs that already cover the code, then an extension just to `UART.SR` is 'uninteresting' and discarded. To address this issue, whenever execution halts due to reaching the end of a stream that was not extended, in subsequent extension attempts the fuzzer also extends this stream. As an additional optimization, if a stream is consistently exhausted, then larger extensions to this stream are used in future attempts.

3.1.2 Input Trimming

Since the length extension stage always increases the size of inputs, the fuzzer may generate inputs that are longer than necessary. Given two different inputs that cover the same code, the shorter input is often preferable for fuzzing. Shorter inputs have several advantages: they can generally be executed more quickly, enabling the fuzzer to attempt a greater number of mutations within the same amount of time. Performing mutations on shorter inputs is more likely to yield a meaningful change to the program's behavior. Additionally, shorter inputs tend to be simpler and thus easier to understand, which aids in the analysis and triaging of crashes.

To address this, MULTIFUZZ utilizes a smart trimming strategy, which consists of two steps: first, the fuzzer automatically removes any unused bytes at the end of a stream (*Auto Trim*), then the fuzzer repeatedly attempts to remove slices within each stream, while ignoring *uninteresting* changes to control flow (*Smart Trim*).

Auto Trim. After execution halts due to reaching the end of one of the input streams, it is possible that there are bytes at the end of other streams that were never accessed. By maintaining a cursor for each stream that tracks bytes read, Auto Trim can efficiently truncate input streams, removing all unread data that can never impact the execution path. This is particularly effective later in the fuzzing campaign when a greater number of extensions are attempted.

Smart Trim. It is also desirable for the fuzzer to remove bytes *within* a stream. For example, consider the control-flow graph (CFG) depicted in Figure 4. This CFG corresponds to a function where the firmware repeatedly reads from the UART status register (`dev->SR`) until a ready bit is set. If the status register stream is filled with bytes where the ready bit is not set, then the fuzzer will execute unnecessary iterations of the loop. However, the trimming strategies employed by traditional fuzzers such as AFL [51] and AFL++ [18] are incapable of removing these bytes primarily because of the

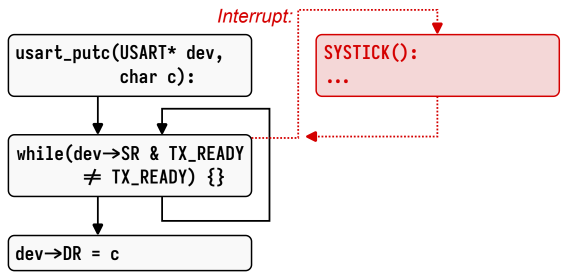

Interrupt:
usart_putc(USART* dev, char c):
SYSTICK():
...
while(dev→SR & TX_READY ≠ TX_READY) {}
dev→DR = c

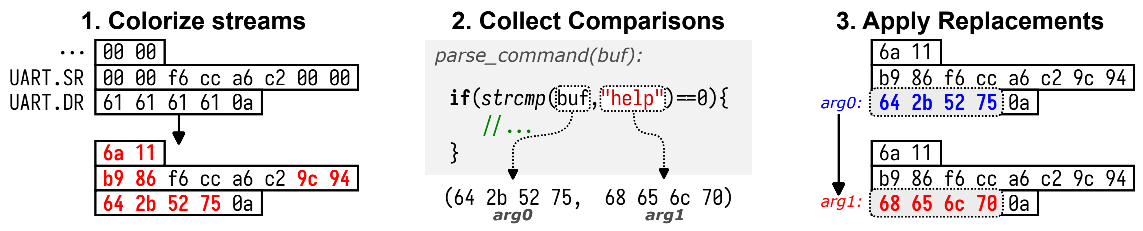
Figure 4: The control-flow graph associated with a function for writing a character to a USART peripheral. In the example, while waiting for the peripheral to be ready, an interrupt occurs, resulting in additional control flow.

criteria they employ to determine valid removals. Traditional fuzzers only consider a removal to be valid if the execution path is unaltered after the removal. Changes in the execution path are detected by monitoring whether different bits are set in the coverage bitmap. For the example in Figure 4, removing bytes without the ready bit set leads to the removal of the loop back-edge and any interrupt handlers that were triggered during the loop. This change to the execution path causes existing trimming strategies to reject the removal.

To improve MULTIFUZZ's ability to trim input streams, we developed an alternative heuristic for rejecting inputs modified during the trim stage, which we call *Smart Trim*. In contrast to the approach used by AFL and AFL++, our approach only rejects removals that prevent *newly* covered code from being reached. This approach is based on the observation that the newly covered code is the only part of the execution path that we care about keeping. In other words, if the trimmed input was found first, the untrimmed input would never have been saved. In Figure 4 for example, if the fuzzer has saved at least one prior input that covers the loop edge and any interrupt handlers, the trim stage can fully remove all non-ready values from the status register (`SR`) stream. Smart Trim can remove bytes from both within and at the end of streams, but it carries a runtime cost proportional to the size of the input. Auto Trim reduces the overhead of Smart Trim by enabling it to start with an already reduced input.

3.2 Multi-stream Input-to-State Replacement

A fuzzer's ability to reach deeper parts of the code in a target can be hindered by complex comparison operations, such as multi-byte integer and string comparisons. This challenge has led to the development of various techniques targeted at addressing this issue [7, 9, 17, 21, 27]. Notably, the input-to-state (I2S) replacement technique introduced by Redqueen [2] has proven highly effective at directly solving many complex comparisons and has been incorporated into several fuzzing frameworks [8, 18, 19, 31]. I2S replacement operates in two stages: first, the fuzzer adds instrumentation to record compar-



```
...      00 00
UART.SR  00 00 f6 cc a6 c2 00 00
UART.DR  61 61 61 61 0a

         6a 11
         b9 86 f6 cc a6 c2 9c 94
         64 2b 52 75 0a
```

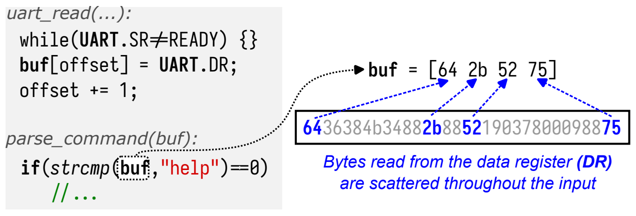
```
parse_command(buf):
  if(strcmp(buf,"help")==0){
    //...
  }

(64 2b 52 75,   68 65 6c 70)
```

```
      6a 11
      b9 86 f6 cc a6 c2 9c 94
arg0: 64 2b 52 75 0a

      6a 11
      b9 86 f6 cc a6 c2 9c 94
arg1: 68 65 6c 70 0a
```

Figure 5: An overview of the multi-stream input-to-state replacement strategy employed by MULTIFUZZ. (**1**) The fuzzer first attempts to *colorize* each stream individually. (**2**) Next the colorized input is executed with instrumentation that logs the operands comparison operations and functions. (**3**) The fuzzer then searches and replaces matching operands within each stream.


Bytes read from the data register (DR) are scattered throughout the input

```
uart_read(...):
  while(UART.SR≠READY) {}
  buf[offset] = UART.DR;
  offset += 1;

parse_command(buf):
  if(strcmp(buf,"help")==0)
    //...
```

```
buf = [64 2b 52 75]
6436384b34882b8852190378000988 75
```

Figure 6: With a file-based input representation, bytes read from the data register of the UART peripheral and stored into the buffer `buf` become scattered throughout the input.

ison operands during program execution, and then the fuzzer searches for bytes in the input that match one of the operands. When a match is found, the fuzzer attempts to replace the input bytes with the correct value (i.e., the matched operand).

In firmware, most complex comparisons involve bytes read from data registers. However, between reading each byte of data, firmware typically waits for new data to become available, either by *polling* the associated status register or by waiting for an interrupt to be triggered. This is problematic when using a file-based input representation as the interspersed accesses between data registers and other peripherals makes it difficult to identify complete multi-byte values within the original input, as shown in Figure 6. In contrast, our multi-stream input representation inherently solves this issue. By separating each peripheral into individual streams, we ensure that the data associated with each stream remains contiguous. This enables MULTIFUZZ to solve complex comparisons by performing I2S-style replacements on individual streams with minimal changes, as shown in Figure 5.

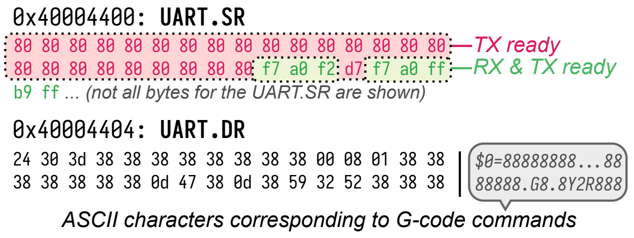
Similar to the original implementation in Redqueen [2], we first attempt to replace non-critical bytes (i.e., bytes that do not affect control flow) with randomly generated values in a process called *colorization*. This increases the entropy of the input, which decreases the likelihood of a spurious match for a comparison. Colorization is performed on a per-stream basis, and we also keep track of streams where every byte has been replaced with a random value (fully colorized) for later use in stream selection (Section 3.3.1).

After colorization, MULTIFUZZ executes the firmware with the modified input and records the operands of each comparison that is encountered. After execution, the fuzzer searches each stream for a match with one of the recorded operands, and if a match is identified, the fuzzer replaces the operand with the other recorded value. To evaluate each replacement, MULTIFUZZ re-executes the firmware with the modified input and checks for changes to the execution path. The replacements are tested individually, with the bytes that were replaced in the input restored to their original values after each attempt. The fuzzer saves any input that reaches new code for future mutations and records any replacement value that changes the execution path into a *per-stream* dictionary. This dictionary is later employed by the length extension stage (Section 3.1.1) and for havoc-style mutations discussed in the next section.

3.3 Multi-stream Mutations

By construction, length extension only modifies values that appear at the end of input streams. However, it is also often beneficial to mutate earlier bytes within the input. To achieve this, MULTIFUZZ includes a havoc stage that randomly applies one or more mutations to any part of the input stream. We then enhance the standard havoc operations by introducing several modifications aimed at improving their effectiveness in mutating multi-stream inputs. We describe these modifications in the remainder of this section.

Stream-to-stream splicing. Splicing involves combining parts of existing inputs to generate new inputs, enabling a fuzzer to repurpose useful sections of previous inputs in different ways. However, data that is useful for one MMIO register is often less relevant for other registers, particularly if they differ in type. For example, Figure 7 shows the streams associated with a UART peripheral from an input discovered while fuzzing the *CNC* target. In the stream for the status register, nearly all bytes have either the transmit ready bit or the receive ready bit set (bits 5 & 6). Conversely, the stream for the data register consists almost entirely of printable ASCII characters. Therefore, using bytes from one stream as part of a splice operation on a stream of a different type is unlikely to be effective. A multi-stream representation enables us to implement a more targeted splicing operation and prevent inter-mixing of data across streams. Whenever a stream is selected for a splice operation, the fuzzer ensures that the operation uses data from streams associated with the same MMIO register (see Section 4). For example, in Figure 7, data from the `UART.DR` stream of one input will be spliced with data from the `UART.DR` stream of another input.

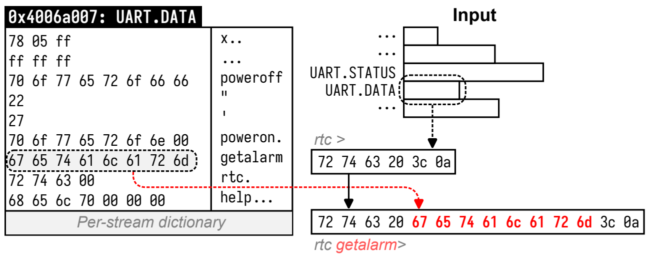


```
0x40004400: UART.SR
80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80 80 f7 a0 f2 d7 f7 a0 ff
b9 ff ... (not all bytes for the UART.SR are shown)

0x40004404: UART.DR
24 30 3d 38 38 38 38 38 38 38 38 00 08 01 38 38
38 38 38 38 38 0d 47 38 0d 38 59 32 52 38 38 38
```

Figure 7: A fragment of an input generated while fuzzing the *CNC* target. This fragment demonstrates the difference between data generated for a status register (`UART.SR`), favouring repetition of bytes for signalling TX/RX ready, compared to a data register (`UART.DR`), favoring ASCII characters.

Per-stream dictionary replacement. As part of the input-to-state stage, the fuzzer collects values found in comparison operations into a per-stream dictionary. During the mutational stage, the fuzzer may mutate an input by copying values from the dictionary into the associated stream. Like stream-to-stream splicing, the per-stream nature of the dictionary ensures that it contains a higher proportion of values relevant to the target stream. In Figure 8, the fuzzer inserts a value of "getalarm" into the UART data register stream using the associated per-stream dictionary.



```
0x4006a007: UART.DATA
78 05 ff                  x..
ff ff ff                  ...
70 6f 77 65 72 6f 66 66   poweroff
22                        "
27                        '
70 6f 77 65 72 6f 6e 00   poweron.
67 65 74 61 6c 61 72 6d   getalarm
72 74 63 00               rtc.
68 65 6c 70 00 00 00      help...
```

```
rtc >
72 74 63 20 3c 0a

72 74 63 20 67 65 74 61 6c 61 72 6d 3c 0a
rtc getalarm>
```

Figure 8: An example of a dictionary mutation when fuzzing the *Console* binary. The fuzzer inserts the dictionary value "getalarm" into the stream for the `UART.DATA` register.

3.3.1 Stream Selection

MultiFuzz utilizes a stream selection strategy that avoids excess mutations to uninteresting input streams. When performing havoc-style mutations, first the fuzzer randomly chooses a stream, then the fuzzer selects a random number of mutations to apply to the stream. The fuzzer selects short streams just as frequently as long streams. Given that longer streams contain more bytes, this effectively biases mutations such that an individual byte within a longer stream is *less* likely to be mutated than an individual byte within a shorter stream. This bias is both *intentional* and *desirable*. Status registers are almost always read before accessing data registers, usually to ensure that the peripheral is ready before performing an action or to check for errors. Consequently, significantly more data is read from status registers compared to data registers. If each byte was given equal weight for mutation, it would result in a disproportionate amount of mutation effort being devoted to status registers, which are typically significantly less interesting from a security perspective. By selecting the stream for mutation *independently* to the location within the stream to mutate, we ensure that the fuzzer better balances its mutational effort.

One issue with this approach is that firmware may access peripheral registers during initialization, which have no impact on the behavior of the firmware. For instance, interactions with configuration registers during initialization have no impact on the behavior of the firmware since the configuration is ignored by the emulator. To reduce the number of mutations performed on these streams, MultiFuzz employs a simple heuristic based on information collected during the colorization (see Section 3.2). During colorization, we track how much of the stream can be replaced by random bytes without changing the execution path, which we refer to as the colorization rate. If a stream has a colorization rate of 100%, then it was never observed to change the execution path, making it unlikely that mutating the stream is useful. However, in rare cases, this heuristic may fail, such as when the value is used in a single comparison with a hard-to-solve value. Therefore, we still occasionally mutate fully colorized streams (with a probability of $0.1\times$ that of mutating other streams).

4 Implementation

To execute the ARM firmware targets, we utilize the timer and NVIC implementation from Fuzzware [42], and ICICLE [8] for emulation and instrumentation. Additionally, we adopt the same three hang detection heuristics as Fuzzware: i) an upper limit on the number of interrupts triggered (excluding SysTick), ii) a limit on the number of blocks executed without any MMIO access, and iii) an overall limit on the total blocks executed. While MultiFuzz is compatible with additional features from Fuzzware (such as MMIO models), the additional features are not used in our evaluation.

For input generation, MultiFuzz's core fuzzing loop follows a similar design to AFL and AFL++ [18, 51], then incorporates the new mutation and extension operations described in Section 3. Additional implementation details are discussed in the following section.

MMIO dispatching. Each multi-stream input is represented as a mapping from a memory address to a stream (an array of bytes with a cursor). Like other firmware fuzzers, MULTIFUZZ assumes that the emulator is aware of the location of the memory region corresponding to memory-mapped Input/Output (MMIO). Whenever firmware accesses memory within the MMIO region, the emulator uses the target address of the access as a key in the map maintained for the input. If no mapping exists for the address, execution is terminated. Otherwise, the emulator returns the next value from the stream to the firmware and updates the cursor. The fuzzer dispatches MMIO accesses of different sizes that target the same address to the same stream. Overlapping accesses that start at *different* addresses are treated as different streams. This typically only occurs as part of APIs that end up discarding the overlap using a bit mask. As a concrete example, when the firmware attempts to read a value from the address `0x40004404`, the emulator checks that `0x40004404` is within the ARM MMIO region (it is) and resolves the value by looking up the address `0x40004404` in the map for the multi-stream input.

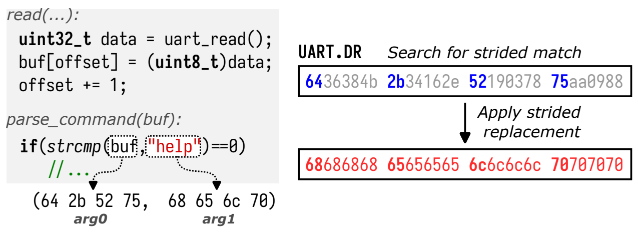
Input-to-state (I2S) replacement. I2S replacement consists of three parts: colorization, comparison logging, and replacement. We re-implemented colorization by adapting the algorithm described in Redqueen [2] to be performed one stream at a time. To capture comparison operands, we use the implementation of CmpLog from Icicle, which injects code at translation time to record the operands of comparison-like instructions and function calls. The operands are stored in a region of memory shared with the fuzzer and used as part of the replacement step. We implement a per-stream replacement process, which includes a restriction on the number of attempts per stream, to avoid excess overhead in the presence of large streams, along with an additional enhancement to address oversized reads, described below.

It is common for firmware to perform oversized reads of MMIO registers and either truncate or mask the value before using it. This causes the values in the original input stream to be padded with the bits removed during truncation, resulting in the interesting bytes being strided within the stream. In cases where the firmware immediately truncates the value after it is read, as shown in Figure 9, we can detect and replace the oversized read with a smaller read when the pattern is encountered by the emulator. To handle cases where truncation is not performed immediately, we also search each stream for strided matches of operands whenever multi-byte reads to a peripheral register are detected. When performing a replacement on a strided match, we repeat the character up to the stride amount, which addresses any alignment issues. An example of a strided replacement being applied is shown in Figure 10.

Hangs from length extension. During testing, we observed that the length extension mutation strategy can *overshoot* and cause the input to reach (and become stuck in) an error handler or another kind of loop. For example, in the *GPS Tracker* binary, triggering the USB ISR (interrupt service routine) could cause the firmware to hang depending on the value read from a peripheral[4]. These states are considered hangs by the fuzzer, and the inputs that reach these states are typically excluded from future mutations. However, if these states are easily reachable, this can result in a significant proportion of extensions leading to hangs. To mitigate this issue, unlike AFL-derived fuzzers, which discard hanging inputs, MULTIFUZZ saves and mutates (but does not attempt to extend) hanging inputs if they reach new coverage.

```
char usart_getc(void) {          ldr   r3, [->UART]
  int value = UART.DR;           ldr   r0, [r3, #0x4]
  return (char)value;            uxtb  r0, r0
}                                bx    lr
```

Figure 9: The source code (left), and assembly code (right) used for reading a byte from a serial device. The code reads 32-bits, but immediately discards the top 24-bits. MULTIFUZZ detects this pattern and resizes the load instruction.



```
read(...):
  uint32_t data = uart_read();
  buf[offset] = (uint8_t)data;
  offset += 1;

parse_command(buf):
  if(strcmp(buf,"help")==0)
    // ...

 (64 2b 52 75,  68 65 6c 70)
```

arg0
arg1
UART.DR
Search for strided match
6436384b 2b34162e 52190378 75aa0988
Apply strided replacement
68686868 65656565 6c6c6c6c 70707070

Figure 10: An example illustrating how checks for strided matches enables matching comparison operations even in the presence of oversized reads from the data register.

Code-coverage feedback. MULTIFUZZ uses block-only coverage for feedback instead of using edge hit counts like existing firmware fuzzers. This approach simultaneously addresses two issues. First, block-only coverage inherently avoids the excess edges added on interrupt entry and exit which can cause the fuzzer to waste time exploring equivalent paths, a problem highlighted by Ember-IO [13]. Second, because interrupts are triggered in a round-robin manner after a fixed number of blocks are executed, our length extension strategy is easily able to reach more interrupt triggers by extending input streams by larger and larger amounts. Consequently, inputs that hit edges multiple times are considered to have unique coverage when coverage feedback is based on edge hit counts[5]. This results in the fuzzer saving huge numbers of increasingly larger inputs, which is avoided when using block-only coverage.

One drawback of block-only coverage is that it provides less feedback for code that needs to be repeatedly executed,

[4]This corresponds to a hardware error on a real device.

[5]Although most approaches bucket hit counts to small power-of-two to mitigate this problem, the large number of interrupts and edges within each interrupt, easily overcomes the power-of-two barrier.

such as loops. However, since our length extension strategy is already effective at exploring loops, this does not inhibit the effectiveness of MULTIFUZZ.

Stage scheduling. The input-to-state stage is performed once for each saved input before any extension or havoc mutations are attempted. Then, the first time an input is selected, the fuzzer favors length extension 90% of the time (to accommodate inputs with new MMIO accesses). In later attempts, the fuzzer randomly selects between the length extension and the havoc stage. Whenever a generated input leads to new code coverage, it is trimmed with Auto Trim followed by Smart Trim before being saved.

5 Evaluation

We designed our evaluation of MULTIFUZZ to answer the following questions:

1. How effective is MULTIFUZZ at fuzzing a wide range of firmware targets and hardware? (Section 5.1 and Section 5.2)
2. How does MULTIFUZZ perform relative to previous state-of-the-art firmware fuzzers? (Section 5.2)
3. How effective are the fuzzing strategies enabled by a multi-stream input representation? (Section 5.3)
4. Can MULTIFUZZ find previously undiscovered bugs in real-world firmware? (Section 5.4)

All the experiments were conducted on an AMD Ryzen Threadripper 3990X, with one core allocated per fuzzing instance. MULTIFUZZ is always configured to start with no initial inputs (the inputs are generated from an empty seed during the length extension stage); other fuzzers are configured to start with the seeds specified in their original paper.

5.1 Evaluation On Synthetic Unit Tests

To determine whether MULTIFUZZ generalizes to a wide range of firmware and hardware, we configured the fuzzer to run the 46 binaries introduced by P^2IM [16]. Within these 46 binaries P^2IM defines 66[6] unit tests that evaluate the fuzzer's ability to reach specific locations in each binary. Within 10 minutes MULTIFUZZ successfully passes 100% of the unit tests, matching the performance of Fuzzware [42] and surpassing both P^2IM and μEmu [54]. This result demonstrates that MULTIFUZZ works across a wide range of firmware targets.

5.2 Comparison with the State of the Art

In this section, we compare MULTIFUZZ's fuzzing performance (in terms of block coverage and bug discovery) against two comparable, fully automated state-of-the-art firmware fuzzers: Fuzzware [42] and Ember-IO [13].

[6] Fuzzware [42] observed that 4 of the original 70 unit tests were invalid, and one binary has no valid unit tests so is excluded.

We use the set of 20 real-world firmware samples used for benchmarking in prior work [13, 42, 54][7]. This includes firmware from a diverse range of applications, OS libraries, and hardware platforms; 10 from P^2IM [16], 6 from μEmu [54], 2 from Pretender [24], and one each from HALucinator [10] and WYCINWYC [38].

We also evaluated three additional firmware targets built on RIOT-OS: *gnrc_networking*, *File system* and *CCN-Lite Relay*. We selected RIOT-OS for its widespread usage, and the three specific targets were chosen to evaluate distinct RIOT subsystems (i.e., the filesystem, the networking stack, and the CCN-Lite library). The *gnrc_networking* and *File system* firmware were compiled to target a STM32F3 MCU and *CCN-Lite Relay* was compiled to target a nRF52832 MCU. Each target was compiled using their default settings without any modifications, and we configured them in a similar manner to the other benchmark binaries, except to allow the triggering of nested interrupts, which is required because the RIOT-OS scheduler runs in handler mode[8]. Each target is fuzzed for 5 trials for 24 hours each, and coverage over time achieved by each fuzzer is shown in Figure 11

In terms of coverage, MULTIFUZZ achieves performance equal to or better than existing techniques across almost all the binaries in the benchmark (statistical analysis is provided in Appendix Table 2). In 13 out of the 23 targets, MULTIFUZZ's *worst* performing trial achieves higher coverage than both Fuzzware and Ember-IO. Across all binaries, MULTIFUZZ is observed to discover new code earlier. This demonstrates that MULTIFUZZ can initialize peripherals more effectively than past approaches due to our multi-stream length extension strategy. This is particularly apparent in several of the binaries including, the *Drone* target, which we analyze in Section 5.3, and the *3D Printer* target where Fuzzware gets stuck in initialization because it fails to generate enough data for the UART peripheral while printing boot messages. For the *Thermostat* target, MULTIFUZZ achieves less coverage, however after further analysis we conclude that the additional code is only reachable using a bug exploit (see Section 5.2.1).

We also evaluated MULTIFUZZ's ability to discover bugs by manually triaging any crashes that were found. To triage crashes, we begin by identifying the root cause issue using a debugger (GDB). Then, to validate the feasibility of reaching the state where the error occurs, we refer to the datasheet corresponding to the microcontroller the firmware was compiled for. For example, we ensure that the data returned from peripherals leading to the state is possible to generate, and that the peripherals are enabled when the corresponding interrupts are triggered. Detailed information about each crash, including the crash-triggering input and a comprehensive root-cause

[7] We only fuzz one of 6LoWPAN Sender/Receiver since they are almost identical (they differ by just 3 blocks), and both crash with the same inputs.

[8] Ember-IO does support not nested interrupts so we only compare against Fuzzware on these targets.

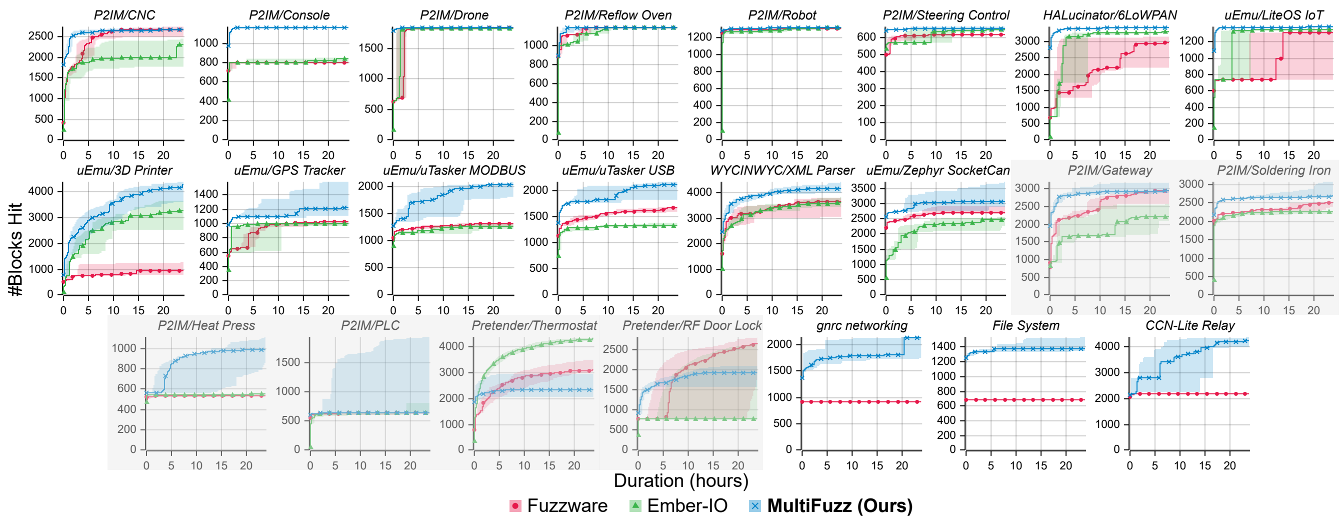


Figure 11: Code coverage for 5 trials of each target over 24 hour period. The solid line represents the median number of unique basic blocks hit, and the area represents the number of blocks hit in the best/worst trials. Subplots shaded in grey represent binaries containing bugs that a fuzzer manages to exploit to reach additional coverage, making the block count metric unreliable.

analysis, is available in our GitHub repository[9]. After identifying the root causes of the crashes found by MULTIFUZZ, we compared the bugs to the ones reported in prior work. We find all 35 unique crashes reported by Fuzzware [42], and the 6 additional crashes found by Ember-IO [13]. Further, MULTIFUZZ finds 13 *new* bugs on the existing benchmark binaries, and 6 bugs in the new binaries, as detailed in Appendix Table 3. On the new binaries, MULTIFUZZ discovered 5 additional bugs (1 in *gnrc networking*, and 4 in *CCN-Lite Relay*). Only one of these bugs could be found by Fuzzware (the reentrancy issue in CCN-Lite Relay)[10].

5.2.1 Bug Exploits Leading to Additional Coverage

Several of the binaries contain bugs that allow arbitrary code execution, enabling the fuzzer to generate inputs that reach code by exploiting these bugs. This can artificially inflate the coverage results for these binaries. Prior work [13, 42] attempts to mitigate this issue by excluding crashing inputs and filtering the coverage results to include only known valid basic blocks (determined using static analysis tools). However, this filtering process is insufficient because a bug exploit can lead to the execution of valid code that is normally unreachable via normal execution.

After analyzing the traces from prior work, we observed bug exploits consistently occurring in both the *Thermostat* and *RF Door Lock* binaries, and occasionally in the *PLC* binary. For example, the *Thermostat* binary contains a bug where the return address of the `get_new_temp` function can be manipulated with fuzzer controlled data by overflowing a stack allocated buffer. By exploiting this bug, the fuzzer can generate inputs that execute code that is normally unreachable. This can include handler functions associated with disabled interrupts, as well as normally unreachable error handlers and print statements. Since bug exploits can lead to a significant increase in the coverage reported by the fuzzer, in our results, the binaries where a bug exploit is detected are shaded in grey.

In the results presented in Figure 11, at least one fuzzer manages to exploit a bug in 6 of the 23 binaries that results in an increase to the number of blocks hit. To detect bug exploits, we analyze the root cause of each bug and add hooks for checking when the bug occurs. For example, in the *Thermostat* target, we inject a hook that checks (by inspecting register values) writes to the stack-allocated buffer do not overflow into the stack slot containing the function return address. If a block is only reachable by inputs that are detected by the hook, then we treat the target as reaching code because of a bug exploit. Additionally, during testing, we observed rare cases where a bug exploit led to increased coverage in both *uTasker USB* and *Zephyr SocketCan*; however, these bug exploits were not observed in the final results.

5.3 Effectiveness of Multi-Stream Techniques

To evaluate the effectiveness of the techniques described in the paper, we conducted 4 additional experiments on each of the firmware targets with MULTIFUZZ configured with subset of the techniques enabled. For configurations where the dynamic dictionary is disabled, only the predefined interesting values (like AFL-style fuzzers) are used for dictionary/value mutations and extensions.

[9] https://github.com/MultiFuzz/MultiFuzz

[10] Fuzzware contains an emulation bug related to the interaction between delayed interrupt checking and instruction barriers. This causes the CCN-Lite Relay target to be emulated incorrectly, complicating bug analysis. However, the reentrancy bug appears to occur prior to encountering the emulation bug.

We start with a basic configuration where only the multi-stream length extension stage (Section 3.1.1) is enabled, then gradually add more complex features. Notably, not all targets require every feature of the fuzzer to fuzz effectively. For example, since the Robot target contains very little code after initialization, the later mutation strategies are not needed to reach all the code in the target. For each target we evaluate whether each addition has a statistically significant impact to the number of blocks reached by the fuzzer in 5 trials. The results of our experiments are shown in Table 1, and we further analyze the impact of the techniques in this section.

Length Extension Only. Our length extension strategy is designed to rapidly discover initial inputs that make it through firmware initialization, serving as an effective starting point for subsequent mutations. As a baseline, we find that length extension proves highly effective, on average reaching 77% of the coverage achieved by the full version of MULTIFUZZ.

As a case study, we analyzed the effectiveness of this configuration on the *Drone* target. By analyzing individual trials (see Appendix Figure 12), we found Fuzzware struggles to progress through 3 functions executed during initialization, each of which involves a significant number of MMIO accesses as part of I2C communication with various subsystems. This requires generating sufficiently large inputs to provide the necessary data for communication. Because Fuzzware does not directly increase the length of inputs, it can take up to 3 hours to complete initialization, while MULTIFUZZ with length extension accomplishes this within minutes.

+Havoc. We then add the multi-stream havoc mutations described in Section 3.3 to the base configuration. The havoc stage enables mutating more than just the end of inputs. Consequently, we see an improvement to the targets that involve logic that depends on different parts of the input. The largest increase occurs in the *Gateway* target which we analyzed (see **Appendix** Figure 13) and determined that the mutations enable the fuzzer to reach several message handlers that length extension alone is unable to reach.

+Trim. In this variant inputs are trimmed using the strategies described in Section 3.1.2 before being added to the input queue. With larger inputs, the fuzzer is less likely to mutate any individual byte, therefore by trimming inputs we make havoc-style mutations more consistent and effective. There are large increases to coverage reached for the *CNC* and *XML Parser* targets, both of which involve parsing structured data where smaller inputs are more likely to be valid data. Trimming also improves the fuzzer's consistency at generating interesting control messages for the *Gateway* target (see **Appendix** Figure 13). We also observed a small decrease in coverage in the *Robot* target, corresponding to code that handles timeouts, that requiring very large inputs to reach. *Heat Press* and *Thermostat* also have large increases, however these can be attributed to bug exploits (Section 5.2.1).

+Input-to-state. In this configuration we combined the previous techniques with a stage that performs input-to-state replacement as described in Section 3.2. MULTIFUZZ achieves substantially higher coverage on binaries that include a shell-like interface (the RIOT-OS binaries, the two uTasker binaries and *Zephyr SocketCan*), as input-to-state allows valid commands and parameters (as analyzed in **Appendix** Figure 14) to be generated more easily. It also provides significant improvements to other targets involving comparisons, including the *3D Printer* target, which uses string comparisons for parsing G-code serial commands, and *Steering Control* which waits for "steer" and "motor" events.

+DynamicDictionary. This configuration represents the full version of MULTIFUZZ, which includes the per-stream dynamic dictionaries described in Section 3.2. Since the input-to-state stage already solves most comparisons, adding a dictionary has a muted effect. The dynamic dictionary is most beneficial for reaching code that requires interactions between multiple strings. For example, in the *File System* target, reaching parts of the code requires first emitting a command to create a file, then a second command that interacts with the same file. However, even with dynamic dictionaries MULTIFUZZ is not always able to find inputs that reach these cases in every trial, reducing statistical significance.

5.4 Analysis of newly discovered bugs

In this section, we analyze a subset of the bugs newly discovered by MULTIFUZZ, focusing on the bugs with a higher security risk. Notably, most of the binaries have been extensively fuzzed by prior work and MULTIFUZZ *still* manages to find 18 new bugs. A summary of all the newly discovered bugs can be found in Appendix Table 3.

Gateway. The firmware processes messages consisting of a command ID and payload. The length of the message is calculated by counting the bytes received up to a termination character. By sending just a termination character, it is possible to send zero-length messages. However, the firmware incorrectly assumes that a command ID is always present, and because the message buffer is reused and not cleared between messages, the command ID read is the value set by the previous message. An integer underflow then occurs when the payload length is adjusted to account for the command ID, eventually resulting in a buffer overflow. Stream-to-stream splicing enhances MULTIFUZZ's ability find this bug by enabling multiple valid messages to be generated more easily.

6LoWPAN Sender/Receiver. The firmware receives 6LoWPAN packets from a SPI bus, storing the packet into a buffer. The firmware then decodes the packet which potentially involves reassembling fragmented payloads. To support reassembly, the packet header includes a fragment offset field. However, the firmware fails to check that the fragment offset is within bounds when copying the payload, causing corruption. MULTIFUZZ's ability to quickly start generating valid

Table 1: Block coverage (median of 5 trials) after 24 hours for different configurations of MULTIFUZZ. Features are added from left to right (e.g., '+Trim' includes: length extension, havoc and the trim stage), the '+DynamicDictionary' column is the full version of MULTIFUZZ. The percentages show the change in coverage compared to the previous configuration. Changes <0.1% are not displayed and statistically significant changes are marked in bold (based on a Mann-Whitney U test with a 0.05 significance threshold). Shaded rows include results where at least one configuration reaches additional blocks via a bug exploit.

Firmware	Extend	+Havoc			+Trim			+Input-to-State			+DynamicDictionary		
				p-value			*p-value*			*p-value*			*p-value*
CNC	2022	1995	(-1%)	0.676	2690	**(+35%)**	0.037	2672	(-1%)	0.835	2672		1.000
Console	807	807		1.000	807		0.600	1132	**(+40%)**	0.009	1167	**(+3%)**	0.019
Drone	1856	1855	(-0.1%)	0.753	1849	(-0.3%)	0.387	1850	(+0.1%)	0.334	1848	(-0.1%)	0.668
Reflow Oven	1191	1192	(+0.1%)	0.434	1193	(+0.1%)	0.389	1192	(-0.1%)	0.651	1193	(+0.1%)	0.651
Robot	1345	1398	**(+4%)**	0.010	1369	**(-2%)**	0.009	1325	(-3%)	0.295	1317	(-1%)	0.395
Steering Control	613	611		0.512	620	**(+1%)**	0.021	655	**(+6%)**	0.011	652	(-0.5%)	0.737
6LoWPAN	3329	3369	**(+1%)**	0.022	3422	**(+2%)**	0.012	3433	(+0.3%)	0.530	3421	(-0.3%)	0.401
LiteOS IoT	1353	1352	(-0.1%)	0.738	1354	(+0.1%)	0.057	1373	**(+1%)**	0.024	1383	**(+1%)**	0.009
3D Printer	1634	1857	(+14%)	0.403	1389	(-25%)	0.095	3794	**(+173%)**	0.012	4268	(+12%)	0.144
GPS Tracker	998	1015	**(+2%)**	0.047	1026	(+1%)	0.172	1093	**(+7%)**	0.011	1218	(+11%)	0.057
uTasker MOD.	1332	1322	(-1%)	0.675	1340	(+1%)	0.210	1560	**(+16%)**	0.012	2028	**(+30%)**	0.021
uTasker USB	1623	1631	(+0.5%)	0.403	1812	**(+11%)**	0.037	1939	**(+7%)**	0.028	2113	(+9%)	0.144
XML Parser	3356	3263	(-3%)	0.676	3946	**(+21%)**	0.012	4163	**(+5%)**	0.037	4164		0.676
Zephyr Socket.	2707	2746	(+1%)	0.296	2786	(+1%)	0.295	3100	**(+11%)**	0.012	3147	(+2%)	0.713
Gateway	2075	2575	**(+24%)**	0.037	2915	(+13%)	0.095	2943	(+1%)	1.000	2936	(-0.2%)	0.676
Soldering Iron	2592	2593		0.835	2631	(+1%)	0.059	2937	**(+12%)**	0.036	2703	(-8%)	0.296
Heat Press	568	568		0.743	927	**(+63%)**	0.012	924	(-0.3%)	0.835	986	(+7%)	0.835
PLC	626	639	(+2%)	0.095	644	(+1%)	0.834	661	(+3%)	0.462	644	(-3%)	0.530
Thermostat	1467	1671	**(+14%)**	0.022	2420	**(+45%)**	0.020	2081	(-14%)	0.270	2352	(+13%)	0.144
RF Door Lock	779	779		0.424	780	**(+0.1%)**	0.020	1957	**(+151%)**	0.010	1935	(-1%)	0.835
gnrc networking	915	923	(+1%)	0.671	925	(+0.2%)	0.060	1831	**(+98%)**	0.010	2128	(+16%)	1.000
File System	846	850	(+0.5%)	0.043	889	(+5%)	0.205	1264	**(+42%)**	0.012	1376	(+9%)	0.095
CCN-Lite Relay	2144	2146	(+0.1%)	0.119	2147		0.025	4070	**(+90%)**	0.012	4235	(+4%)	0.835

packets allows it to spend longer fuzzing the 6lowpan interface enabling it to find this bug. Further analysis revealed the bug is attributable to a known vulnerability in Contiki-OS, which provides the networking interface used by the binary.

uTasker_USB. In one of the bugs within this target, the firmware fails to validate that the interface index used for device/interface requests from the setup packet is in-bounds. For example, when handling a `SET_LINE_CODING` message, if the target interface index is too large, then the settings from the message payload will be copied to arbitrary memory.

gnrc_networking. The code responsible for creating an IPv6 echo packet calculates the total size of the packet by adding the requested payload size to the header size. For a large payload size, an integer overflow occurs when computing the total size. This leads to a packet smaller than the header size. If the overflown value is equal to zero the code dereferences a NULL pointer, for other values smaller than the header size, the code overflows a buffer when filling the packet payload.

6 Discussion

Direct Memory Access (DMA). For MULTIFUZZ we focused on the input-generation aspect of the fuzzing process. We use a similar approach to existing fuzzers such as Fuzzware [42] and Ember-IO [13] when dealing with DMA. MULTIFUZZ does not differentiate between DMA and other MMIO accesses and instead relies on external configuration when necessary. For the binaries included in the benchmarks, we use the configuration files from Fuzzware. However, our approach is fully compatible with other approaches that improve the emulator's ability to handle DMA peripherals, such as DICE [35] or SEmu [55]. Data associated with DMA peripherals could be represented as separate streams in MULTIFUZZ and mutated using operations specialized for DMA.

False positive crashes. In addition to the crashes caused by the bugs found by MULTIFUZZ, we also identified 7 additional false positives, and after further analysis we reclassified 9 of the bugs reported by prior work also as false positives. Most of the false positive crashes are related to interrupts being triggered *before* the firmware fully initializes data related to managing the peripheral (prior work reported these issues as "Unchecked Init"). However, on real ARM MCUs, an interrupt for a peripheral should only be triggered if the bit associated with the interrupt is enabled in the NVIC (nested vectored interrupt controller) *and* the peripheral is enabled and configured to generate the relevant interrupt. For MULTIFUZZ we use the same approach as Fuzzware [42] to handle interrupt triggering, where interrupts can be scheduled as soon as they are enabled in the NVIC. However, firmware may configure the NVIC *before* fully initializing and enabling peripherals, which can result in false positive crashes.

We also found 3 additional false positive crashes related to hardware assumptions. Two of these occurred in *Zephyr Sock-*

etCAN related to the 'canbus attach/detach' sub-commands, causing buffer overflows. The third was related to clock rate initialization in *6LoWPAN Sender* leading to unbounded recursion resulting in a stack overflow. A comprehensive analysis of false-positive crashes is available in our GitHub repository.

Multi-stream fuzzing with MMIO modeling. We have demonstrated that our multi-stream input representation, when used to implement a firmware fuzzer, outperforms prior approaches, including those that utilize modeling techniques. However, our approach is completely independent to MMIO modeling strategies, allowing MULTIFUZZ to be combined with modeling approaches such as Fuzzware [42].

7 Related work

Fuzzing techniques have demonstrated a proven capability to find bugs in diverse software [25, 36]. As such, substantial research effort has been devoted to improving the effectiveness of fuzzers [2, 7, 17, 32, 39]. For example, Angora [7] attempts to explore different input lengths by tracking the return value for `read`-like function calls. However, these approaches are typically focused on testing software written for desktop operating systems and are not directly applicable to fuzzing firmware. Although a range of approaches are proposed to analyze the security of firmware more broadly, including targeted approaches to search for specific vulnerabilities [22, 50], or those involving human analysis [5, 34, 44, 55], we focus on elaborating upon methods explored for fuzzing firmware.

Multiple approaches have explored firmware fuzzing without the need for direct hardware access. Pretender [24] attempts to re-host firmware in an emulated environment by observing interactions between the firmware and hardware during execution on the original hardware. HALucinator [10] makes the observation that many IoT devices utilize a hardware-abstraction-layer (HAL) instead of interacting with the hardware directly and emulates the HAL instead of the underlying hardware. Concurrently with our work, SAFIREFUZZ [47] improves the efficiency of this approach by statically rewriting the binary to achieve near-native execution on a more powerful ARM host machine instead of relying on emulation. However, HAL-level emulation requires the time-consuming task of identifying and manually implementing replacements for all HAL functions; a task made more difficult if the functions are inlined as a result of compiler optimizations. P^{2}IM [16] attempts to automatically classify the type of peripheral using several heuristics. Fuzzware [42] infers models for peripherals by locally applying symbolic execution to analyze how peripheral data is used. They demonstrated that their approach to modelling reduces the input overhead, significantly improving the effectiveness of fuzzer mutations. Ember-IO [13] foregoes modelling and introduces two techniques, FERMCov to avoid saving excess inputs added on interrupt entry and exit, and peripheral input playback (PIP) to allow the fuzzer to generate values that are repeated. However, all these approaches either focus on making more efficient use of the inputs consumed by the emulator or preventing the fuzzer from exploring unprofitable paths in a program. In contrast MULTIFUZZ focuses on better input generation as part of the fuzzing frontend. The use of alternative input representations to improve fuzzer effectiveness has been explored in domains outside of firmware fuzzing [1, 3, 4, 23, 40, 41, 49]. For example, Nyx [46] represents bytecode programs for fuzzing hypervisors as directed acyclic graphs. Further, generic fuzzing frameworks such as LibAFL [19] are designed to be compatible with multiple input representations. However, none of the existing input representations are suitable for fuzzing firmware.

Notably, the concurrent work, Hoedur [43], also uses multi-stream inputs to fuzz firmware. MULTIFUZZ places a greater emphasis on length extension strategies compared to Hoedur, incorporating specialized extension mechanisms and generally performing larger extensions. To solve string comparisons, Hoedur constructs a shared dictionary from strings found in the firmware using static analysis. This approach limited to constant string comparisons only, in contrast, MULTIFUZZ's input-to-state stage supports both non-constant string comparisons and non-string memory comparisons. Additionally, Hoedur assigns streams to access contexts (a tuple consisting of the program counter and MMIO address). We define streams solely on MMIO addresses to ensure data read from peripheral registers are contiguous even after inlining optimizations, which improves the consistency of input-to-state replacements. SplITS [14] solves string comparisons in single-stream inputs by introducing an iterative algorithm for matching partial strings that is resistant to data scattering and input stability. This approach has a higher runtime cost to input-to-state replacement in multi-stream inputs and does not address the other issues with single stream inputs.

8 Conclusion

MULTIFUZZ utilizes a novel multi-stream input representation for handling MMIO accesses to fuzz firmware. This new representation is combined with an efficient multi-stream length extension strategy, improved trimming approach, and multi-stream specific mutation operations enables MULTIFUZZ to achieve higher coverage and discover more bugs than existing state-of-the-art firmware fuzzers.

Acknowledgements

The work was supported by the Cyber Security Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centres Programme, and through the Australian Government's Research Training Program Scholarship (RTPS).

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed Systems Security Symposium*, 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*, USENIX Security, 2023.

[4] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, USENIX Security, 2019.

[5] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference*, ACSAC, 2020.

[6] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Network and Distributed System Security Symposium*, NDSS, 2016.

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

[8] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. Icicle: A Re-Designed Emulator for Grey-Box Firmware Fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-Box Concolic Testing on Binary Code. In *IEEE/ACM International Conference on Software Engineering*, ICSE, 2019.

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

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

[12] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing Embedded Systems Using Debug Interfaces. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[13] Guy Farrelly, Michael Chesser, and Damith C. Ranasinghe. Ember-IO: Effective Firmware Fuzzing with Model-Free Memory Mapped IO. In *ACM ASIA Conference on Computer and Communications Security*, AsiaCCS, 2023.

[14] Guy Farrelly, Paul Quirk, Salil S. Kanhere, Seyit Camtepe, and Damith C. Ranasinghe. SplITS: Split Input-to-State Mapping for Effective Firmware Fuzzing. In *European Symposium on Research in Computer Security*, ESORICS, 2023.

[15] 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 Asia Conference on Computer and Communications Security*, AsiaCCS, 2021.

[16] Bo Feng, Alejandro Mera, and Long Lu. P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *USENIX Security Symposium*, USENIX Security, 2020.

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2020.

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

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

[20] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. Dissecting American Fuzzy Lop: A FuzzBench Evaluation. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–26, 2023.

[21] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium*, USENIX Security, 2020.

[22] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images. In *IEEE Symposium on Security and Privacy*, SP, 2022.

[23] Samuel Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. *Department of Informatics, Karlsruhe Institute of Technology*, 2018.

[24] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *International Symposium on Research in Attacks, Intrusions and Defenses*, RAID, 2019.

[25] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3), 2020.

[26] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed Selection for Successful Fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2021.

[27] Intel. Circumventing Fuzzing Roadblocks with Compiler Transformations, 2016.

[28] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *USENIX Security Symposium*, USENIX Security, 2020.

[29] 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*, USENIX Security, 2021.

[30] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Annual Computer Security Applications Conference*, ACSAC, 2020.

[31] AWS Labs. Snapcharge: Lightweight Fuzzing of a Memory Snapshot using KVM, 2023.

[32] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. In *International Conference on Software Engineering*, ICSE, 2023.

[33] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2022.

[34] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec, 2020.

[35] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *IEEE Symposium on Security and Privacy*, SP, 2021.

[36] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: an open fuzzer benchmarking platform and service. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2021.

[37] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar^2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research*, BAR, 2018.

[38] 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.

[39] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium*, USENIX Security, 2020.

[40] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[41] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level Fuzzing. In *USENIX Security Symposium*, USENIX Security, 2021.

[42] 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*, USENIX Security, 2022.

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

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

[45] 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.

[46] 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*, USENIX Security, 2021.

[47] Lukas Seidel, Dominik Maier, and Marius Muench. Forming Faster Firmware Fuzzers. In *USENIX Security Symposium*, USENIX Security, 2023.

[48] Jack Tang and Moony Li. Project Triforce: Run AFL on Everything! *Black Hat Europe*, 2016.

[49] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium*, USENIX Security, 2022.

[50] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2020.

[51] Michal Zalewski. American Fuzzy Lop: A Security-oriented Fuzzer, 2010.

[52] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, USENIX Security, 2019.

[53] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient Greybox Fuzzing of Applications in Linux-Based IoT Devices via Enhanced User-Mode Emulation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2022.

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

[55] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2022.

A Appendix

A.1 Block Coverage Statistical Analysis

Table 2: Block coverage after 24 hours for Fuzzware (FW), Ember-IO and MultiFuzz (median of 5 trials). To test for statistical significance we conducted Mann-Whitney U tests comparing the results for MultiFuzz to Fuzzware and Ember-IO, and report the associated *p*-values in the final two columns. Entries in bold represent the best performing fuzzer with statistical significance (0.05 significance threshold). The metrics for entries shaded in grey may be unreliable due to code reached via bug exploits.

Firmware	**Average blocks hit**			***p-value***	
	FW	Ember	MultiFuzz	vs FW	vs Ember
CNC	2672	2303	2672	0.90	0.02
Console	805	843	**1167**	0.01	0.01
Drone	1836	1835	**1848**	0.01	0.01
Reflow Oven	1192	1192	1193	0.65	0.52
Robot	1306	1315	1317	0.06	0.11
Steering Control	613	644	**652**	0.01	0.01
6LoWPAN	2967	3289	**3421**	0.01	0.01
LiteOS IoT	1319	1348	**1383**	0.01	0.01
3D Printer	933	3259	**4268**	0.01	0.02
GPS Tracker	1025	1001	**1218**	0.01	0.01
uTasker MODBUS	1313	1252	**2028**	0.01	0.01
uTasker USB	1667	1336	**2113**	0.01	0.01
XML Parser	3641	3629	**4164**	0.01	0.01
Zephyr SocketCan	2718	2468	**3147**	0.02	0.02
Gateway	2941	2214	2936	0.53	0.01
Soldering Iron	2515	2265	**2703**	0.01	0.01
Heat Press	539	554	**986**	0.01	0.01
PLC	637	642	644	0.02	0.25
Thermostat	3116	**4284**	2352	0.01	0.01
RF Door Lock	2661	782	1935	0.09	0.13
gnrc networking	926	**-**	**2128**	0.01	**-**
File System	686	-	**1376**	0.01	-
CCN-Lite Relay	2211	-	**4235**	0.01	-

A.2 MultiFuzz Component Impact Analysis

To support Section 5.3, we plot the time required by different fuzzer configurations to generate inputs covering specific code regions in various binaries. The right subfigures plot

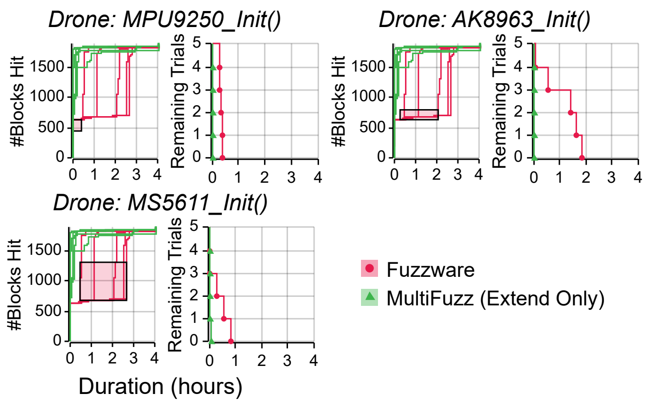


Figure 12: Time required to complete various initialization functions in the *Drone* target.

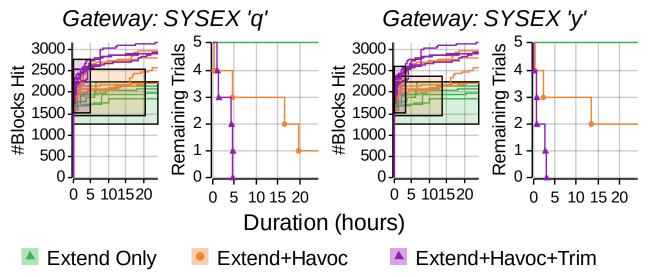


Figure 13: Time required to generate specific control messages in the *Gateway* target.

the amount time each fuzzing trial is stuck at a particular region of code after reaching it for the first time. e.g., the *Drone: AK8963_Init()* plot indicates that two Fuzzware trials cover the function in less than an hour, while the remaining trials take longer. The left subfigures are provided as context to show where within the overall coverage graph the each code snippet is encountered. The shaded rectangular region represents the blocks hit while solving the code fragment.

Figure 12 analyzes the impact of length extension on rapidly progressing through initialization compared to an existing state-of-the-art fuzzer (Fuzzware [42]). It can take Fuzzware almost 3 hours in total to finish initialization in some trials, while MULTIFUZZ with length extension alone completes initialization within minutes.

Figure 13 explore the necessity of havoc-style mutations in reaching various control message handlers in the *Gateway* target. Control messages consist of a command ID framed with start and stop bytes. Without havoc-style mutations (i.e., Extend Only), the fuzzer *never* manages to generate a message with a valid command ID. Introducing havoc-style mutations enables the fuzzer to mutate the command ID while leaving the start and stop bytes alone, albeit inconsistently. Trimming inputs significantly increases the speed and consistency of generating control messages, allowing the configuration with trim enabled to generate both target messages within 5 hours.

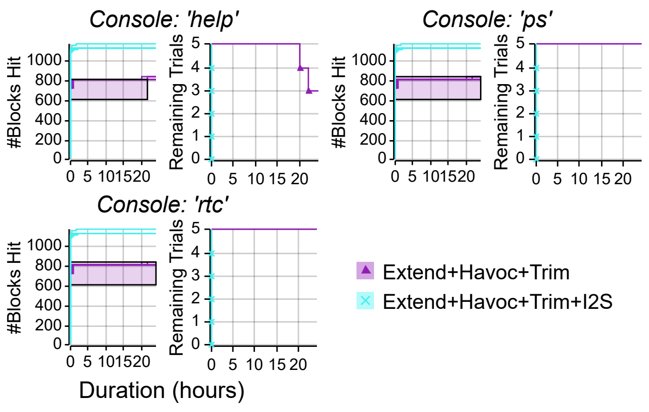


Figure 14: Time required to reach command handler functions in the *Console* target.

Figure 13 investigates the impact of input-to-state (I2S) replacements. Without a mechanism to solve string comparisons, the MULTIFUZZ struggles to reach any of the command handlers. Across all trials, the configuration without I2S only manages to reach a command handler twice. With I2S, all command handlers are reached within minutes of fuzzing.

A.3 Responsible Disclosure

We disclosed security related bugs by sending bug-reports to vendors/developers in accordance with the security policies listed for the associated project. All reported issues have either been fixed, or based on feedback from maintainers, are not considered to be security critical.

Table 3: Summary of the 18 new bugs found by MULTIFUZZ.

Target	Description	Status
6LoWPAN	Fragment offset is not bounds-checked	Fixed upstream at the time of reporting.
Zephyr SocketCan	net pkt command de-references a user provided pointer.	Fixed at the time of reporting.
Zephyr SocketCan	canbus config command fail to validate argument count.	Code rewritten in latest release.
Zephyr SocketCan	canbus sub-commands fail to verify device type.	Noted as a known API limitation.
Zephyr SocketCan	pwm sub-commands fail to verify device type.	Noted as a known API limitation.
Gateway	Incorrect handling of zero length sysex messages.	Reported & Fixed.
utasker MODBUS	Direct manipulation of memory using I/O menu.	Intentional feature for debugging.
utasker USB	Direct manipulation of memory using I/O menu.	Intentional feature for debugging.
utasker USB	Out-of-bounds access from interface index.	Reported & Acknowledged.
GPSTracker	strtok not checked for NULL in gsm_get_imei.	Input is trusted, not reported.
GPSTracker	strstr not checked for NULL in sms_check.	Input is trusted, not reported.
GPSTracker	strstr not checked for NULL in gsm_get_time.	Input is trusted, not reported.
GPSTracker	strtok not checked for NULL in gsm_get_time.	Input is trusted, not reported.
gnrc_networking	Integer overflow when computing echo packet size.	Reported & Fixed.
CCN-Lite Relay	Data race during stdio initialization.	Reported & Fixed.
CCN-Lite Relay	Use-after-free of temporary interface.	Reported & Acknowledged.
CCN-Lite Relay	Re-initialization of shared global variables.	Reported & Acknowledged.
CCN-Lite Relay	Incorrect handling of content messages with invalid URIs.	Reported & Acknowledged.