FTZ: A State-Inferring Fuzzer for the TCP/IP Stack of Zephyr

Valentin Huber

Master's Thesis at

Zürich University of Applied Sciences ZHAW
Institute of Computer Science InIT
Information Security Research Group

In collaboration with Cyber-Defence Campus

Advised by
Dr. Peter Heinrich, Zürich University of Applied Sciences
Damian Pfammatter, Cyber-Defence Campus

February 22, 2025

Abstract

Fuzzing has proven to be an effective technique for finding software errors. However, applying it to complex targets such as the TCP/IP stack of the real-time operating system Zephyr remains challenging because of its deep OS integration, highly structured input data, and state-dependent behavior.

This thesis presents FTZ, an open-source state-inferring fuzzer targeting the TCP/IP stack of Zephyr. Built on LibAFL, FTZ uses NativeSimulator alongside a custom shared memory-based network driver to run Zephyr efficiently. It supports modeling and mutating network packets as binary data or parsed structures and two strategies for targeting input parts to mutate. To further improve performance, FTZ uses a heuristic based on received packets to estimate the state of Zephyr, which is then used either directly or mapped to state transitions as feedback to the fuzzer.

Initial experiments show that Zephyr does not behave consistently, with both coverage and state feedback fluctuating. FTZ investigates the source of these inconsistencies and implements additional logic to statistically decrease inconsistency rates. Evaluating the techniques in FTZ shows that modeling packets as parsed structures is more effective, even though it limits the input space the fuzzer can generate. Selecting the last input message to mutate improves performance compared to mutating a random packet. Finally, state transition feedback allows FTZ to reach higher code coverage in Zephyr than directly using the state of incoming packets.

This thesis shows the feasibility of performing efficient fuzz testing on Zephyr. It further introduces a novel state-inference heuristic and shows how simple strategies can be employed to significantly increase the performance of a fuzzer for a stateful target with complex input data interdependencies.

Keywords: Protocol Fuzzing, Stateful Fuzzing, Zephyr, LibAFL.

Contents

1 Introduction

Zephyr is an open-source real-time operating system (OS) hosted by the Linux Foundation [1]. Real-time OSs allow programmers to define time constraints on code sections, which are then guaranteed by the process scheduler. Adhering to these constraints is particularly important for safety-critical systems where delays in computation can pose a serious risk, such as control of a robot. For the compiler to ensure this, all operations must be verifiably complete within a constant maximum time or fail safely. [2]. Zephyr is employed in a wide range of products, including large industrial equipment such as wind turbines, MedTech devices like hearing aids, laptop controllers, and embedded devices such as emergency water detectors [3].

Zephyr projects typically connect to other devices or the internet using one of the OS' networking libraries, such as the Zephyr TCP/IP stack. As many of these devices require internet access, any defect in the network stack is comparatively likely to be remotely exploitable and may propagate from device to device, and is thus critical [4]. Combined with the potentially dangerous devices Zephyr runs on, this makes the correctness of these parts of Zephyr arguably important [5]. Recent research further found that errors within a network stack are concentrated in the transport and network layer [6].

Fuzzing has been established as a widely used technique [7] to find software defects by repeatedly running a particular program under test (PUT) with a wide range of inputs and observing the execution for illegal program states such as crashes. It has been applied to great effect: Microsoft requires dominant software to be fuzzed prior to release [7], and its fuzzer SAGE [8] "reportedly found a third of the Windows 7 bugs between 2007-2009" [9]. Google mentioned finding 20,000 vulnerabilities in Chrome alone using fuzz testing [7], while its OSS-Fuzz program continuously fuzzing open-source software has resulted in over 6,500 vulnerabilities and 21,000 functional bugs fixed across 500 projects deemed critical enough to be included in OSS-Fuzz [10].

Ever since its inception in the nominal work by Miller *et al.* [11], who used random data to test Unix utilities, many improvements to fuzzing have been introduced to the state of the art. Among many others (Google Scholar retrieves more than 60,000 results in a search for "fuzz testing" [12]) include observing the program execution to guide the fuzzer when selecting the next input to test or improved oracles such as memory sanitizers, which introduce additional runtime checks for illegal actions such as out-of-bounds memory accesses. Section 2.1 provides additional background on the specific fuzzing technique employed in this project called mutational fuzzing.

However, fuzzing the network stack of an operating system poses three additional challenges: (1) they are deeply integrated with the OS, (2) they require highly structured input data, and (3) they have internal state. These challenges are further elaborated in Section 2.2.

1.1 Research Questions

For this thesis, I developed a proof-of-concept Fuzzer targeting the TCP/IP stack of Zephyr (FTZ), built on top of the fuzzing library LibAFL. FTZ uses two different input representations, appropriate mutation techniques, two strategies for targeting input parts to mutate, stateful feedback based on a heuristic requiring no target instrumentation, and reliability-enhancing logic to test Zephyr effectively. This thesis attempts to answer the following Research Questions:

Building a fuzzer to target the TCP/IP stack of Zephyr

RQ 1a How can Zephyr be run as a fuzzing target with as few changes as possible?

RQ 1b What are the efficiency and reliability challenges in doing so?

RQ 1c How can Zephyr be instrumented to enhance a fuzzer?

State-inferring heuristic

RQ 2a How can state be inferred from a TCP/IP server without additional instrumentation?

RQ 2b What are the efficacy implications of using this state information as feedback to the fuzzer either directly or by estimating state transitions?

Input modeling and mutation

RQ 3 What are the efficacy implications of modeling network packets as binary data compared to parsed data, along with appropriate mutation techniques?

Selection of input part to mutate

RQ 4 What are the efficacy implications of targeting a random packet in the input to mutate next compared to consistently mutating the last packet of the input?

2 Background

This section presents additional background information required to understand the remainder of this thesis.

2.1 Mutational Fuzzing

FTZ implements mutational fuzzing. This is a technique used across a wide range of influential fuzzers, such as AFL++ [13], syskaller [14], libFuzzer [15], Honggfuzz [16], or OSS-Fuzz [17]. A mutational fuzzer maintains a corpus of inputs, first seeded at startup with a number of entries that are either generated randomly or loaded from a predefined set. The fuzzer then iteratively chooses one of the inputs in its corpus, mutates it, and evaluates it in the PUT [18].

During the execution, mutational fuzzers observe the PUT in certain ways, such as by collecting information about which parts of the PUT's code were executed. These fuzzers are also known as greybox fuzzers — a midpoint between blackbox fuzzers, which do not know anything about the internals of the PUT during the execution, and whitebox fuzzers, which employ heavy-weighted analysis tools on the target, such as symbolic execution or taint analysis [7].

The data observed during target execution is then used to decide if the current input should be added back into the corpus. The most common form of such feedback is coverage-guided fuzzing, which adds an input to the corpus if new coverage is measured in the PUT. This incentivizes the fuzzer to explore previously untested parts of the PUT [13].

2.2 Fuzzing Network Stacks Is Hard

As introduced in Section 1, fuzzing an operating system's network stack introduces three challenges.

2.2.1 Deep integration with OS

OS' built-in network stacks are typically deeply integrated with other subsystems, such as the memory management subsystem for accessing the network interface card and socket buffers, or the system's scheduler for event notifications. Extracting the network stack from the rest of the kernel to be run independently is therefore often infeasible since it would require extensive additional engineering. Such reengineering is error-prone and may result in findings that do not hold on the original implementation [19].

To test the network stack of an operating system using a fuzzer, one therefore has to run the entire operating system, which includes a lot of logic unrelated to the network stack itself, thus introducing a significant performance penalty. This further means that any compiler-level sanitizer, such as AddressSanitizer, has to be introduced into the OS' build system, which may not be possible because the additional logic may break assumptions made by the OS developers (see Section 4.1). Finally, OS' kernels are, by definition, tightly integrated with hardware, and running an operating system therefore requires either dedicated hardware (which may be hard to observe and communicate with) or some form of hardware and environment simulation, introducing additional performance penalties and complexity overhead.

2.2.2 Network Packets Are Highly Structured

Table 1 shows the structure of a packet, as it would be used to seed FTZ's corpus. Many fields across all layers depend on the values of other fields in their header or even other layers. Examples include length fields marked with green or checksums marked with blue. Checksums such as the CRC32 frame check sequence in the ethernet trailer or the 16-bit one's compliment-based checksums in the IPv4 and TCP headers are prime examples of logic fuzzers struggle to penetrate. This is because mutating one part of the data requires precise changes to another, which is highly unlikely to happen in random mutation [20]. This challenge has garnered some attention from previous work, such as removing the program sections responsible for checking checksums from the target program [21] or bypassing them [22].

For TCP, these dependencies go even further: the sequence and acknowledgment numbers in the TCP header depend not only on the current packet but also on previously sent packets. Similarly, the TCP header flags have different meaning depending on the state of the recipient's TCP stack.

2.2.3 TCP Stacks Have Internal State

The fact that TCP is a stateful protocol increases the state space to be explored by the fuzzer dramatically, and the fuzzer may find it hard to reach deeper states [23]. Indeed, stateful targets are listed among the remaining challenges in fuzzing by Boehme *et al.* in their review paper [24]. Ba *et al.* demonstrate that simply relying on coverage information is insufficient to thoroughly test stateful code, but the state of the target has to be taken into account [25]. A similar conclusion was drawn during the evaluation of StateFuzz [26].

Layer	Field	Size
Ethernet	Destination MAC	6 bytes
	Source MAC	6 bytes
	802.1Q VLAN Tag	4 bytes
	EtherType	2 bytes
IPv4	Version	4 bit
	Internet Header Length	4 bit
	DSCP & ECN	1 byte
	Total Length	2 bytes
	Identification	2 bytes
	Flags & Fragment Offset	2 bytes
	Time To Live	1 byte
	Protocol	1 byte
	Header Checksum	2 bytes
	Source IP	4 bytes
	Destination IP	4 bytes
	Options	variable
TCP	Source Port	2 bytes
	Destination Port	2 bytes
	Sequence Number	4 bytes
	Acknowledgment Number	4 bytes
	Data Offset & Reserved	1 byte
	Flags	1 byte
	Window	2 bytes
	Checksum	2 bytes
	Urgent Pointer	2 bytes
	Options	variable
Payload		variable
Ethernet	Frame Check Sequence	4 bytes

Table 1: Structure of a TCP/IP packet encapsulated in an ethernet frame, as used during seeding of FTZ. Gray fields are optional, green fields are length-dependent, and blue fields are checksums.

The inherent statefulness of the TCP protocol introduces two fundamental challenges to the architecture of the fuzzer:

1. In order to maintain the ability to triage the software error identified by the fuzzer, one must be capable of repeatedly inducing the same error. Since the PUT may be put into different states with each message, all previous interaction with a PUT instance need to be recorded, records which may grow increasingly large across a fuzzing campaign with billions of executions and even more packets exchanged. Additionally, even recordings across long interaction chains may not suffice if the target does not always behave predictably. To mitigate this challenge, FTZ restarts Zephyr before each input evaluation, trading off the additional runtime for increased simplicity and reliability and reduced memory and storage consumption.

2. The input to be mutated needs to contain all previous interaction with the PUT, which put it into a specific state, in addition to the packet the fuzzer attempts to trigger an error with. Section 4.4 presents different approaches evaluated in this project.

Daniele *et al.* introduce a taxonomy of stateful fuzzing in which they define a stateful system as "a system that takes a sequence of messages as input, producing outputs along the way, and where each input may result in an internal state change" [23]. TCP stacks are such a stateful system. This thesis will follow the naming convention outlined in their same work, reserving "the term message or input message for the individual input that the System Under Test (SUT) consumes at each step and the term trace for a sequence of such messages that make up the entire input." [23]

2.3 LibAFL

LibAFL is a fuzzing library written in Rust by the maintainers of the popular greybox fuzzer AFL++. They have observed how many different fuzzers implement similar patterns and data structures — or even fork an existing fuzzer — but only change one specific part of the fuzzer. However, these improvements rarely get implemented in the upstream fuzzer. This leaves the fuzzing community with a list of provably effective algorithms implemented in incompatible projects. LibAFL provides generic interfaces for common functionality such as schedulers or mutators and basic implementation of each.

Additionally, many advanced algorithms such as the mutation scheduler from MOpt [27], the weighted input scheduler from AFL++ [13], Hitcount coverage postprocessing from AFL [28], or execution-dependent mutation from REDQUEEN [22] have been added. In 2022, when LibAFL was released, improvements from 20 prior works were implemented and evaluated individually and in combination. Since then, additional algorithms and improvements have been introduced to make LibAFL a powerful and flexible base for FTZ. [29] Section 4.6 presents the details of how LibAFL is used in FTZ.

3 Related Works

As this thesis explores testing a TCP/IP stack of a real-time OS using state feedback, two sets of topics relate to it: projects performing fuzz testing on Zephyr and other embedded OSs, and projects performing fuzz testing on implementations of stateful protocols.

3.1 Fuzzing Embedded Operating Systems

Recently, fuzz testing of embedded devices has received significant attention [30]. However, the primary focus of the research community with regard to fuzzing embedded systems has been on generic methodologies to run their OSs at all, with the main challenges being differences in instruction sets and environment interactions. Thus, they rely on emulation layers and provide various manually or automatically generated hardware or memory-mapped IO (MMIO) models [30].

Common emulators used by fuzzers targeting embedded systems include Quick Emulator (QEMU) [5], [31], [32], Unicorn [4], [33], or Ghidra Emulator [34], [35]. Environment and hardware modeling techniques include high-level hooking, recording hardware-in-the-loop interactions, manual heuristics, symbolic execution, fuzzing, and extraction from user manuals [30].

As Zephyr provides built-in functionality in `native_sim` to run the OS as a native executable on a Linux host (see Section 4.1), FTZ does not require any emulation. Furthermore, `native_sim` provides the required mapping of hardware from host to Zephyr, and FTZ introduces custom logic to exchange networking packets between the fuzzer and PUT, as described in Section 4.3. This negates the need for additional environment modeling. It is worth noting that Zephyr provides a pseudo-target similar to `native_sim`, which allows running it in QEMU. Because NativeSimulator, which provides the basis for `native_sim`, was specifically built to be faster than its emulation counterpart [36], this was not pursued further. However, there exist examples of fuzzers using Zephyr's QEMU such as para-rehosting [37].

While some of these improvements are evaluated against the network stack of embedded OSs — some even against Zephyr — they usually rely on off-the-shelf coverage-guided fuzzers like libFuzzer or AFL, with no additional consideration for the state of the target or format of network packets specifically. An example of this is VP-CFG [38], which uses libFuzzer to mutate IP packets passed to Zephyr using the provided Serial Line Internet Protocol (SLIP) network driver. Herdt and Drechsler evaluate their virtual prototype on Zephyr's TCP/IP stack using coverage-guided fuzzing. Based on the arguments from Sections 2.2.2 and 2.2.3, this approach is unlikely to penetrate the stateful parts of the network stack logic, such as the TCP logic, past the initial packet parsing code.

Since FTZ does not rely on re-hosting or introduces environment modeling techniques, advancements in this field are out of scope for this thesis. Refer to the various dedicated review papers [30], [40]–[42] for additional details on generic techniques to fuzz embedded OSs. However, certain projects introduce improvements to input modeling and mutation, which are relevant to FTZ:

- Fuzzware [4] uses MMIO interactions as its input and extracts structural information about it using symbolic execution. It then uses a binary data input model, which is mutated using AFL's and AFL++'s default random mutation, which is then mapped to structured MMIO inputs and sequentially fed to the PUT.
- ES-Fuzz [43] extends Fuzzware by introducing dynamic MMIO models interpreting binary data to generate PUT inputs. These add constraints on the mapping output based on selective dynamic symbolic execution to infer state transitions in the PUT, which change over the course of an execution according to the changing PUT state.
- SplITS [33] enhances Fuzzware by observing non-contiguous string comparisons in the tested firmware and uses this information to guide mutation.
- MultiFuzz [35] uses independent input streams for each MMIO peripheral register, which are automatically extended when they run out of data

to prevent the PUT from waiting on data. It further records comparisons with register values in the PUT and keeps a dictionary of interesting values for each input stream to improve its mutation.

- Hoedur [44] uses unique input streams for each unique combination of address counter, register address, and access size, dynamically creating and extending streams as needed. This allows using strongly typed data for each stream, thus improving mutation.

While these projects are able to infer parts of the input structure of network packets, they remain unable to compute more complex data inter-dependencies such as checksums. In addition to these academic works, there are reports of non-academic fuzzing campaigns available:

Zephyr has an integration to test parts of the software that are callable from user code using libFuzzer. Compilation of an instrumented version of the OS and linking with libFuzzer is possible with a simple build system configuration entry. However, the example project used as an explanation in Zephyr's documentation does not contain any kernel logic and instead dummy code showing the abilities of coverage-guided fuzzing [45], [46]. There is also an adaption of this example allowing the use of FuzzBuzz as the fuzzer backing the execution instead of libFuzzer [47].

According to [48], the libFuzzer integration has successfully been used to test Zephyr's Bluetooth stack. While the blog post mentioning the campaign claims it helped find several bugs, it does not provide any details about the campaign, employed techniques, or evaluation. Besides targeting a different part of Zephyr, it uses libFuzzer, a pure coverage-guided fuzzer without any state feedback or structure-aware mutation. The blog post further describes using the Renode hardware systems simulator in combination with AFL [28] and AFL++ [13] to fuzz Zephyr, specifically the `console/echo` sample. This relies on Renode's proprietary emulator and underlies the same limitations as their first project.

3.2 Stateful Protocol Fuzzing

Like fuzzing of embedded OSs, targeting stateful protocol implementation has received considerable attention. A set of works published before October 2024 have been selected to represent the potential improvements implemented and evaluated in FTZ. Refer to the survey papers such as [23], [49], [50] for a more complete overview of the state of the art.

Natella and Pham propose a benchmark for stateful protocol fuzzers in ProFuzzBench [51], containing a suite of 10 protocols and 11 open-source implementations of those to be tested. TCP notably is not part of this benchmark directly but is used as an underlying protocol for others, such as FTP. HTTP status codes provide state feedback — certain protocols, such as FTP, already provide these, while others are patched to do so based on a superficial heuristic for their state. The authors note that configuration of the targets is not taken into account and multi-party (≥ 3 participants) protocols cannot currently be fuzzed. Non-determinism in the programs makes feedback (like code coverage) less predictable and thus fuzzing less performant because it introduces non-differentiable duplicate entries into the corpus.

However, of the fuzzers projects discussed in this thesis, only StateAFL [52] is evaluated against this benchmark, while EPF [53] notes the absence of such a benchmark for their work.

3.2.1 Observing State

As described in Section 2.2.3, a key challenge in fuzzing implementations of participants to stateful protocols is providing information about this state back to the fuzzer. Extracting this information from a PUT has been achieved using various techniques across projects:

Manual Annotation Ijon [54] is an extension to AFL and requires analysts to annotate the PUT's code manually. Ijon introduces logic at the annotated location to add entries to an AFL-style map in either a tainting or counting mode. It can further directly include state information, such as variable values in how the edge coverage is calculated, and store the maximum value a particular variable reaches during execution for the fuzzer to then maximize in the fuzzer. However, the only target Ijon is evaluated on and resembling a protocol is the CROMU_00020 target of the Darpa Cyber Grand Challenge, which requires inferring the state of the target to trigger the error successfully. There, Ijon showed improved performance compared to unmodified AFL.

Automatic Annotation Manual annotation of state variables is tedious, error-prone, and requires a decent understanding of the PUT internal structure from the analyst. Because of this, many projects attempt to use a heuristic to automatically find variable or memory locations representing the current state of the PUT and pass this information back to the fuzzer.

Ba *et al.* rely on the intuition that variables representing the state of their target are often assigned from named constants in an enum struct. Instrumentation is injected at all locations that update these variables to pass the information back to their fuzzer SGFuzz [25].

SandPuppy [55] performs an initial run of the PUT, capturing variable-value traces, and uses a heuristic based on this information and static analysis of the source code of the PUT to identify state-representing variables. These are then instrumented with an Ijon-like feedback mechanism. This process is repeated at certain intervals during the fuzzing campaign. Compared to Ijon, minor improvements were achieved, such as additional solved Super Mario Bros. levels. It further shows that state feedback generally improves achieved coverage compared to pure coverage-guided fuzzers such as AFL, AFL++, or REDQUEEN, and even compared to the heuristic to identify state-representing variables in SGFuzz.

StateFuzz [26] identifies state-representing memory by looking for long-lived variables, updatable by the user, and either change the control flow or are used to index into memory. StateFuzz identified a four-digit number of state variables in the Linux and Qualcomm MSM kernel used in Google's Pixel line of smartphones.

Other Greybox Heuristics StateAFL [52] relies on compile-time probes observing memory allocation and I/O operations to identify state-representing memory locations. State inference is then done based on fuzzy hashing of long-lived memory areas. Their approach proves effective, even compared to other state-aware fuzzers such as AFLNet.

TCP-Fuzz [56] and Ankou [57] take the combination of executed branches as a heuristic for the state of the target. However, this approach leads to a set of states much larger than the states of the abstract TCP state machine and the one powering the server (TCP-Fuzz found 47.9K state transitions in the TCP stack mTCP). Because of this, Ankou further reduces this complexity for a custom adaptive fitness function.

3.2.2 Use of State Information

The state information extracted or inferred using these techniques is then passed to the fuzzer, which uses it in the following ways.

Feedback In mutational fuzzers, state information can be used in several ways to decide whether an input should be considered interesting and thus added to the corpus. Examples of this include StateFuzz [26], which considers inputs interesting that trigger additional, previously unchanged states, state values representing a new value range, or new extreme values. Value ranges are binned values of states determined to be congruent using symbolic execution. AFLNet [58] allows users to specify that not only states but also state transitions should be used to determine the interestingness of an input. DDFuzz [59] uses a similar approach but does not directly rely on state information and instead uses execution of new inter-data dependency relationships as feedback.

Scheduling The state of the PUT, as observed using any of the methods described above, can then be used to improve both input and mutator scheduling, as shown by the following approaches:

Several projects create an abstract data structure representing the state space and possible state transitions of the PUT. This can then be used to improve scheduling. Examples include ZFuzz, which calculates an ad-hoc directed graph based on measured states, while SGFuzz [25] constructs a similar data structure called a state transition tree. Finite state machines are a popular structure to model state transitions, as implemented by Jero *et al.*, whose fuzzer requires manual specification of the FSM based on the protocol specification, or Hsu *et al.* [61], who additionally implement FSM minimization. Autofuzz [62], on the other hand, employs a bioinformatics algorithm to approximate the FSM.

These structures are then used in several ways: AFLNet [58] uses a custom heuristic based on statistics about each state to schedule the next input to be mutated. SGFuzz [25] guides the fuzzer to under-explored parts of the state-space by scheduling input traces leaving the PUT in such a state for the next mutation. The structure can be further used to schedule which mutator(s) should be invoked next [61]. ZFuzz [63] does this based on a formula incorporating state depth, coverage, number of transitions, and number of mutations based on this state.

3.2.3 Target Differences

While all of the projects presented above introduce provably effective approaches to increase a fuzzer's performance on their targets by using state feedback, only TCP-Fuzz [56] has the same target as FTZ: TCP/IP stacks.

Others target BitTorrent [57], DNS [52], FTP [52], [58], [62]–[64], HTTP2 [25], Modbus-TCP [65]–[70], MSNIM [61], RTSP and other media streaming protocols [25], [58], SMTP [52], SSH [52], or SSL/TLS [25], [59], [63] libraries, or SCADA systems [53]. Some of

these fuzzers use TCP/IP to facilitate communication between the fuzzer and PUT, but it is not evaluated independently or declared as out-of-scope. To compare the effectiveness of their improvements, some fuzzers are further evaluated against mazes, levels of the game Super Mario Bros, or file formats with complex data interdependencies [54], [55], [57], [59].

Thus, comparisons are impossible without reimplementing the logic of either related work for FTZ's targets or vice versa. Additionally, such changes to related works would also need to include the custom logic to exchange packets with the PUT introduced in FTZ — this also applies to TCP-Fuzz.

3.3 Input Modeling and Targeted Specific Mutation

Section 2.2.2 introduced how mutation on inputs with high data interdependence, such as length fields or checksums, is one of the core challenges for mutational fuzzers targeting logic processing such inputs. While most protocol fuzzers rely on random mutation, certain works have proposed improvements to better mitigate this issue.

EPF [53] uses population-based simulated annealing to schedule which packet type to add or mutate next in conjunction with coverage feedback. It requires a Scapy-compatible implementation of packet types to be fuzzed to ensure packet structure. TCP-Fuzz models the inputs to the TCP/IP stacks tested as a list of system calls and packets, and mutates these lists while ensuring certain constraints between input parts, such as `socket`, `bind`, `listen` and `accept` being always called in order if a connection is in a particular state. However, TCP-Fuzz randomly selects which part of the input is mutated next. Since FTZ employs a similar technique, it includes the major downside of this approach: the structure of the packets and constraints to be ensured after mutation have to be specified manually.

Alternatively, approaches such as MTA [68], GAN-Fuzz [67] or Autofuzz [62] attempt to automatically learn the structure of the inputs based on generative adversarial networks [67], simplified transformers [68], or bioinformatics techniques [62].

3.4 Machine-in-the-Middle

Unlike other works, FitM [64] and Autofuzz [62] attempt to fuzz both the server and client implementations of a protocol library through machine-in-the-middle mutation. The former uses userspace snapshots of the entire client and server processes instead of a log of previous messages (see Section 2.2.3), while the latter constructs a finite state automaton from the observed packets to mutate messages intelligently.

4 Implementation

This section describes the architecture and implementation details of FTZ. It relies on Zephyr's `sockets/echo` sample, a simple example implementing a TCP/IP server that echos any data back to the client.

4.1 native_sim

During the compilation of Zephyr, a target board has to be set. One such target the Zephyr project provides is called `native_sim`, with which the entire operating system and all user code can be compiled into a Linux executable. It is based on NativeSimulator and provides a wrapper around Zephyr with a main function, a scheduler mapping from the host's scheduler to Zephyr's scheduler, hardware models, interrupt controllers, and basic CPU functionality like threading and start/sleep/interrupt calls [36]. Its integration with Zephyr provides functionality to use the host operating system's functionality, such as Ethernet, UART, or display drivers, among others [71].

The ability to run Zephyr as a native executable allows the use of native debugging tools such as `gdb`. It further makes computationally expensive translation layers such as QEMU unnecessary, thus increasing the efficiency of the target. This is especially relevant for a fuzzer since it runs the target millions of times.

When targeting `native_sim`, Zephyr's build system can further be instructed to compile the code using AddressSanitizer (ASAN). This compiler pass instruments the target binary with additional runtime checks for memory errors such as out-of-bounds accesses to heap, stack, or global variables, use-after-free and use-after-return errors, or double frees [72]. Both the default toolchain based on `gcc` and the alternate LLVM/`clang`-based toolchain provide such instrumentation.

During this project, I discovered a bug in Zephyr's device registration when using the ASAN implementation of clang, which I needed to use for its superior coverage instrumentation (see Section 4.2). The additional instrumentation would change the binary layout, breaking the convention used in Zephyr for registering devices. I worked together with the maintainers of Zephyr to triage and fix this bug [73].

`native_sim` allows manipulating the clock of Zephyr to run the operating system faster or slower than in real-time or disabling any speed restrictions. However, the latter renders any `sleep`-interrupted

```
void __sanitizer_cov_trace_pc_guard_init(
  uint32_t *start,
  uint32_t *stop
) {
  for (uint32_t *x = start; x < stop; x++) {
      *x = 0;
  }
}
void __sanitizer_cov_trace_pc_guard(
  uint32_t *guard
) {
  *guard += 1;
}
```

Listing 1: SanitizerCoverage Hook Implementations

loop into one that runs nearly unrestrictedly fast. This, in turn, leads to near-instant timeouts of connections and is thus unsuitable for interaction with another system, such as the fuzzer. However, passing `--rt-ratio=<n>` to the executable built with the target `native_sim` allows a user to set the ratio of how quickly Zephyr's virtual time passes compared to real-time.

4.2 Coverage Information

To guide the fuzzer and measure its progress, FTZ measures the executed code under each trace (refer to Section 2.1 for a short introduction to the role of coverage feedback). A method commonly used and well-supported by LibAFL to do this is using `clang`'s SanitizerCoverage compiler pass. It instruments the code with two functions, as shown in Listing 1.

The first is called at program startup and allows initializing a memory section (delimited with `start` and `stop`). The second is called on each basic block edge, with a guard unique to this edge, pointing in the previously initialized memory section. The implementation above counts how often every edge is executed.

Initial measurements suggested that the coverage measured when running Zephyr using the `native_sim` wrapper is inconsistent (see Section 5.1), particularly in the code sections responsible for matching host thread to Zephyr thread, the execution of which depends on the (unpredictable) host scheduler. Thus, an alternate implementation of `__sanitizer_cov_trace_pc_guard` is used in FTZ, which only marks the visited edges as executed instead of counting the executions to stabilize the results.

To prevent additional instability, I reduced the changes to existing logic in both the code and build system to a minimum. To achieve this, my implementations for the SanitizerCoverage functions are in a separate module included in the build system. A single entry is added to the appropriate CMake file, and a single option is added to Zephyr's configuration system.

To increase performance and error-resistance, the coverage is directly written into shared memory created by the fuzzer. The shared memory is opened during initialization based on information passed through environment variables. The information needed to map from the SanitizerCoverage-reserved memory section to the shared memory section is stored. The appropriate address in the shared memory section is calculated and marked as visited on each edge. While this introduces a slight performance penalty on each edge due to the address mapping, this approach does not require any postprocessing on the client's part, such as copying the collected information back to the fuzzer. This is particularly important in cases of crashes in the PUT since postprocessing functions can no longer be called.

4.3 Exchanging Network Packets with Zephyr

Zephyr's `native_sim` already provides an ethernet driver for a TAP interface over a zeth interface [71]. However, I opted to build an alternative solution for this project to circumvent the need for additional host kernel interaction. FTZ provides a custom ethernet driver for Zephyr that transmits network packets between the fuzzer and target using shared memory.

As described above, I again wanted to limit the required changes to existing logic as much as possible. To achieve this, the configuration system is extended to provide an alternative to the existing TAP-based ethernet driver. The new configuration is then used in the appropriate CMake files to include the logic for the shared memory-based ethernet driver. Similar to the existing TAP-based driver, the logic is split into a part compiled into the kernel directly and one for all interaction with the host system.

The driver requires a single shared memory segment between the fuzzer and Zephyr. The fuzzer initializes it, and the information required to open it in Zephyr is passed using environment variables. There, it is mapped into the PUT memory in the driver initialization hook. The memory section is split into four parts: one buffer and a status field each for both directions. The status field is set to a negative number as long as the buffer is empty and used as a length field when a packet is sent in either direction. Zephyr's TAP-based driver utilizes sleep-interrupted polling of the network, and so does the shared memory-based

implementation.

Initial experiments showed that certain packets are received at unpredictable intervals or must be consistently sent to access the TCP logic. The responses to these interactions are manually constructed in the fuzzer and sent to Zephyr. Currently, FTZ manually responds to ICMPv6 packets of type NeighborSolicit and RouterSolicit and to ARP requests for the fuzzer's MAC address.

4.4 Input Modeling and Mutation

Section 2.2 introduced multiple challenges in fuzzing network stacks in general and TCP stacks specifically. FTZ implements multiple approaches to mitigate some of those challenges, the details of which are discussed here.

4.4.1 Trace Modeling and Mutation Target

AFLNet [58] and TCP-Fuzz [56] select a random message from the trace to mutate. However, this strategy has one significant intuitive drawback: If the corpus contains a long trace generated across many executions to arrive in a specific state by combining a large number of messages, and the mutation strategy chooses the first message to mutate, the remaining messages are likely going to be irrelevant because the initial state changes.

Alternatively, one could approach this problem in a fashion similar to FitM [64]. Maier *et al.* essentially build a tree by snapshotting executions and only appending to them. This way, any state can still be reached, but it does not require the fuzzer to needlessly evaluate long traces.

In FTZ, I introduce `ListInput`. It contains a list of messages, and each entry can be mutated in two ways: Either the trace is mutated by appending, removing, or replacing a message, or individual messages are mutated. For the former category, FTZ is configured to append messages from either the recorded interaction or generated randomly. Mutators on individual messages can be configured using `ListInput::map_to_mutate_on_last` or `ListInput::map_to_mutate_on_random` to either target the last or a random message, representing the strategies in AFLNet and TCP-Fuzz, and FitM respectively.

4.4.2 Seeding

Since both options of modeling a trace include mutators appending messages to a trace, one could seed the corpus with a single, empty trace. In principle, the fuzzer will still eventually find its way through the entire state space. However, one can speed up this process greatly with an improved seeding strategy. For this, I first traced a complete and fully legal interaction between a client and Zephyr. The trace is then filtered for TCP messages from the client to the server. Then, the corpus is seeded with slices of this filtered list of messages, starting with an empty trace and then including increasingly more consecutive messages, all starting with the first filtered message.

The fuzzer is forced to include *all* traces into its corpus. This is necessary as the server may not respond to every message with an answer, and two traces therefore may look the same to the fuzzer.

This improved seeding strategy does not diminish the state space the fuzzer can explore—the empty trace is still part of the corpus. However, it provides the fuzzer with traces ending in many different legal states, thus negating the need for the fuzzer to find its way to deeper states by pure mutation.

4.4.3 Input Message Modeling and Mutation

Then, there is the question of how to model individual messages. Here, three fundamental approaches could be considered: (1) any representation and exclusive crossover mutations, (2) byte arrays and random mutation, and (3) parsed structure with different levels of manual fixing of values after random mutation.

First, when one only allows crossover mutation of full messages, i.e. appending a message randomly drawn from any other entry in the corpus or a fixed list, how the input is modeled does not matter, since the messages themselves never change. However, this approach limits the kinds of traces a fuzzer is able to generate to permutations of seed messages and is thus unsuitable for general fuzzing.

Second, one could model a packet as a byte array and mutate it with the default `havoc_mutations` from AFL++ [13]. These include, among others, byte increments and decrements, bit flips, crossover copying or inserting of subsections of the byte array, setting a byte (combination) to a magic value known for triggering specific types of errors, or deleting parts of the byte array. However, as discussed in Section 2.2.2, TCP packets are highly structured and include various checksums. These are unlikely to be calculated correctly by random mutation. Thus, fuzzers relying on packets generated through `havoc_mutations` will not be able to explore the server's code past the initial parsing logic.

The last option includes modeling a packet as a data structure representing all parts of the raw packet across all layers. These could then be mutated more selectively. Parts of the packet representing a num-

ber can be mutated with number-specific mutators (like setting them to interesting values), bit fields can be mutated as such, and one can fix magic values or checksums manually. This approach represents a tradeoff between the ability to generate a wide range of messages and thus traces while keeping the generated messages reasonably valid so as not to be outright rejected by the parser.

When employing the third option, one has fine control over what mutations are applied by the fuzzer and what data has to be set manually. One could even allow the fuzzer to mutate checksum or length fields and only selectively override and fix the mutated values to the correct ones. This approach tests all parts of the input parser and is still able to reach the business logic of the TCP stack.

The drawback of this last option is that it requires the analyst to specify the structure of the network packet. For common protocols such as TCP/IP, libraries are available that significantly reduce the required programming overhead. However, one still needs to decide which parts of the message should be randomly mutated and which should be fixed. The engineering overhead when using LibAFL has been significantly reduced with `MappingMutators`, which are discussed in Section 4.7, along with other improvements to LibAFL implemented during this project.

4.5 State Inference Heuristic

Section 2.2.3 discussed how the internal state of the TCP stack is a major challenge for fuzzers. One promising response to mitigate this issue is adding state feedback to the fuzzer. This allows the fuzzer to consider the differences in the state of the PUT's TCP state machine when assessing whether a new trace is worth adding to the corpus.

To implement this, I mapped incoming messages to a state, represented by a number: Any valid TCP packet is mapped to the header field containing the flags as a `u8` representation. Other packets are categorized into different states, representing the alternate packet content (such as packets not containing layer 4 data) or the error source (such as packets with an invalid IP header). These states are then enumerated, resulting in a categorization of incoming packets into a total of 267 unique states.

These state numbers are then used to index into a memory slice representing the TCP state space, similar to how coverage is catalogued in a coverage map. Compared to coverage measurements, one needs to consider that simply counting how often each state is measured incentivizes the fuzzer to produce increasingly long traces, even if no new actual state is reached. FTZ therefore only marks which states were measured at any point during the evaluation of a trace. This further improves reliability, with a similar argument presented in Section 4.2.

Section 4.3 discusses that Zephyr emits certain ICMPv6 packets at unpredictable intervals. To optimize the reliability of state-based feedback, ICMPv6 packets are therefore not mapped to the state memory map.

Two ways of calculating state feedback were implemented in FTZ:

1. Messages emitted from Zephyr are simply mapped to states and stored in the state memory map. This mode will be called state feedback.

2. Alternatively, the state transition is calculated from an incoming packet and the previous packet to incentivize the fuzzer to explore all possible state transitions. To implement this, the fuzzer keeps track of previously sent and received messages — since state transitions depend on packets sent from the client and the server, both directions are considered. On each incoming message, FTZ calculates a unique identifier for the measured state transition between the current and last packet, which is then used to index a larger state-diff map. State transitions are only measured for each incoming message. Otherwise, when the fuzzer finds a way to send at least two arbitrary packets without a response, the entire state space would be enumerated by fuzzer-emitted packets while the TCP stack of Zephyr remains in the same state. Using the state transitions as feedback will be labeled as using state-diff feedback.

4.6 During an Execution: Implementation Details

This section describes the implementation of the different parts of FTZ in detail. For additional details, refer to the architecture diagram of LibAFL in Figure 1 and their paper [29] for the interaction of the different structures.

4.6.1 Central Infrastructure

The fuzzer relies on `CentralizedLauncher` to spawn its clients, each with its data structures and each spawning its own instance of Zephyr. `CentralizedLauncher` synchronizes inputs added to one of the client's corpi to minimize re-execution and combine the progress made across all clients. Two clients are spawned with a different configuration

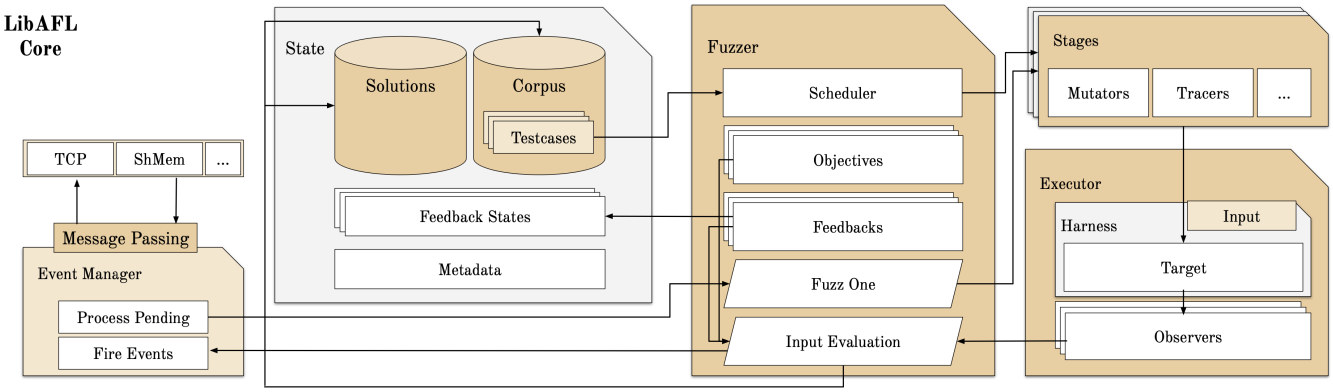

Figure 1: LibAFL's Architecture [29]

compared to all other clients: The central synchronizing client does not receive a mutational stage; its only job is synchronization. The last client receives additional feedbacks with constant interestingness values, which report data about the host system, such as available memory, to the monitor.

Communication between clients is done through Low Level Message Passing (LLMP), a shared memory-based message-passing system provided by LibAFL. The absence of a requirement for kernel-based synchronization primitives such as locks allows FTZ to scale across thousands of clients effectively.

Central monitoring is configurable in FTZ but defaults to an `OnDiskJsonAggregateMonitor`, which will aggregate data across all clients and log it to a machine-readable file. A graphical status board and a monitor relaying a client's and aggregated logs to the global logging system are also provided. LibAFL further supports more complex logging to different data types or other systems, such as a Prometheus instance, which are easily integrated into FTZ.

4.6.2 Client Setup and Operation

At the start of the fuzzing process, each client initializes its data structures. These include the following:

- A shared memory section to measure coverage in.
- A `ConstMapObserver` to handle the coverage, wrapped in a `HitcountsMapObserver` for binning postprocessing.
- A `TimeObserver` to measure execution time.
- A `PacketObserver` responsible for keeping track of transmitted packets, mapping states to a coverage-like map (refer to Section 4.5), and providing all required information for the current execution, such as a has of all packets for deduplication of inputs, the states and state-map for debugging purposes, and a base64-encoded packet capture (pcap) representation of the interaction, to allow graphical tools like Wireshark to be used in debugging.
- A `StdMapObserver` based on the map from the `PacketObserver`, to handle state coverage information.
- Combined feedback containing
  - `MaxMapFeedback`s for both code and state coverage information, the only feedbacks contributing values towards the interestingness of the current input.
  - A `PacketMetadataFeedback` attaching the metadata provided by the `PacketObserver` to the testcase containing the input.
  - A `TimeFeedback`, `InputLenFeedback`, and for the last client host system measuring feedbacks, attaching the execution time, input length, and other information to the metadata.
- Two corpi: a `InMemoryCorpus` for the working corpus of the fuzzer, and an `OnDiskCorpus` for all crashes found.
- A `StdState` containing a random number generator and the corpi.
- The appropriate mutators depending on the selected representation of the trace (see Section 4.4).
- `StdMOptMutator` is used as a mutator scheduler — it implements the algorithm first presented in MOpt [27].

- A `StdMutationalStage` wrapping the mutator scheduler and handling the main mutation workflow.
- A `StdWeightedScheduler` as an input selection scheduler with a power schedule, as known from AFL++ [13].
- A `ReplayingFuzzer` handling the main fuzzing loop — see below for more details.
- A custom `Executor` implementing all the necessary logic to interact with Zephyr — see below for more details.

After the required data structures are set up, the corpus is filled based on a recorded trace (refer to Section 4.4.2). Because some trace subsets result in the same state coverage feedback but may still be useful to the fuzzer as starting points from mutations, they are force-added to the corpus and individually evaluated to enhance them with metadata.

Finally, the main fuzzing loop is started. Here, the fuzzer repeatedly calls the scheduler to select the next corpus entry to be mutated. Then, it calls the mutational stage to repeatedly select how many and which mutations to execute on a copy of the input and calls the evaluator in the fuzzer.

Here, the additional logic in `ReplayingFuzzer` compared to LibAFL's built-in `StdFuzzer` comes into play. It sets up a map to store execution results and then repeatedly does the following:

1. First, the observers are reset.
2. Then, the input is passed to the executor to be run.
3. After that, postprocessing on observers is done.
4. The results from the observer(s), whose results are to be made more reliable, are hashed and combined with the exit status of the execution.
5. This combination is then used to index into the execution count map, where `ReplayingFuzzer` counts how often each result is measured.
6. Then, the measurements are evaluated as follows:
   a. If a crash is observed, the fuzzer returns early from the loop to ensure all crashes are caught by the fuzzer.
   b. If the results observed the most have been measured more than a configurable number more often than any other, and the latest execution results in the most common measurements, the execution results are assumed to be "correct" and returned.
   c. If, after a configurable maximum number of executions, the fuzzer is still unable to get a decisive winner, the evaluation is short-circuited, and the input is not added to either corpus.
7. Finally, the measured ratios are stored and reported to the monitor.

The executor receives the input and starts its operation by marking the new run in the fuzzing client state and resetting the packet transmission buffers. Then, Zephyr is spawned in a subprocess with the shared memory information for the coverage and network buffer as environment variables. FTZ then waits for Zephyr to start while logging packets emitted by the PUT. Packets requiring a manual response are responded to, as described in Section 4.3.

After this interaction, the trace is converted to a list of raw byte vectors to be transmitted to Zephyr. One by one, the packets are sent to Zephyr and added to the `PacketObserver`. In between sending packets, FTZ waits for Zephyr to respond. Since there is currently no way to wait until Zephyr is done processing an incoming packet, FTZ waits until a set time has expired since the last outgoing or incoming packet. During this phase, manual responses are emitted if appropriate. After the last packet was sent and no packet was received for the set timeout, the subprocess running Zephyr is shut down and checked for a crash to be reported.

There exists a balance between the inter-packet wait time and Zephyr's `--rt-ratio` value (see Section 4.1) to maximize Zephyr's consistency. If Zephyr is run too quickly, and the inter-packet wait time is set too long, the connection times out before the second packet is sent. In the opposite case, Zephyr may not be done processing a packet before the fuzzer attempts to send the next packet. Initial experiments showed the sweet spot to be at a `--rt-ratio` of 1 (thus running Zephyr according to the host's real-time) and an inter-packet wait time of 100ms, interrupted 5 times to check for incoming packets.

4.6.3 Helper Functionality

FTZ contains additional logic helpful during development and triage of found issues. First, an implementation of the shared memory-based ethernet driver for the userland network client `smoltcp` allows manual interaction with Zephyr through shared memory. Additionally, a set of postprocessing scripts are available, which, among other things, extract pcap files from the corpus and produce plots as shown in Section 5.

4.7 Contributions to LibAFL

During this project, I have contributed a range of improvements implemented for FTZ as generic version to LibAFL across more than 25 pull requests, including more than 10,000 changed lines of code. These changes include:

- `MappingMutator`s are essential for any project using compound input types — they allow mapping mutators targeting a certain type to those where the initial input type can be extracted from using custom logic. One such example would be applying `havoc_mutations` to the payload field of the TCP packet.[1]
- `CentralizedLauncherBuilder::overcommit` and `LauncherBuilder::overcommit` allows users to launch multiple fuzzing clients per CPU core. This is useful in specific projects like FTZ, where an individual client does not fully load a CPU core because of required wait states, for example, for synchronization.
- `int_mutators` include the applicable mutators from `havoc_mutations` targeting numeric types.
- `BoolMutator`s flip boolean values.
- `ValueInput`, an improvement to how inputs encapsulating a simple data type are implemented.
- A set of macros for mapping and combining mutator lists and their types.
- Improved flexibility for certain schedulers to work on any observer.
- `OnDiskJsonAggregateMonitor` logs the data aggregated across all fuzzing clients at a certain interval to a file containing machine-readable data.
- Several general architectural and clean-code improvements.
- A generic adaptation of `ListInput`, including additional mutators and support for the existing `MultipartInput` — a simplified variant of `ListInput` with additional keys for each message.
- Several bugfixes related to `MultipartInput`.
- `StdFuzzer::with_bloom_input_filter`, an improvement to `StdFuzzer` which uses a probabilistic filter to reduce repeated execution of the same input[2]

[1] Currently, only non-crossover mutations are supported in FTZ, because of a limitation in LibAFL with using nested `MappingMutator`s. This is required to map mutators from their raw target type (such as byte array or number) to the parsed input structure and then again to the composite type such as `ListInput`.

5 Evaluation and Results

FTZ was evaluated along multiple axes, starting with experiments on the consistency of the target and scalability of the fuzzer and then evaluating the effectiveness of the different improvements implemented in FTZ.

All experiments were performed on a 64-core AMD EPYC-Milan machine, using the configuration and parameters described in Section 4 unless indicated otherwise.

5.1 Consistency

To evaluate the consistency of the PUT when executed by the fuzzer, I changed FTZ to use `StdScheduler`, which selects inputs from the corpus at random, to generate a broader range of inputs. I further set up `ReplayingFuzzer` to evaluate input traces until the most common appears at least 100 times and a factor of 1.5 more often than any other input, with a limit of 1500 executions per input.

When FTZ is set up to use both coverage and state-diff feedback, and requiring inputs to be consistent in both their coverage and state measurements, the results in Figure 2 are measured. The results for runs checking coverage and state-diff feedback exclusively can be found in Figures 3 and 4. All figures show the average stability measured across trace lengths. The evaluations were run across 64 cores, with 10 clients per core, for approximately 24 hours and 40 million target executions on individual feedbacks, and 48 hours for the combined feedback.

To find the source of the different coverage maps measured across multiple executions of the same input, I added additional logging to `__sanitizer_cov_trace_pc_guard` (refer to Section 4.2), recording the guard and the address the function returns to. The latter is used to map guard to source code location. At the same time, in the fuzzer, the differences between executions are mapped back to the guard and recorded. This information is then combined and sorted by how often each guard appears. The most common guards appearing in this list correspond to the following functions:

1. `k_work_init_delayable` from `kernel/work.c`

[2] Not currently used in FTZ.

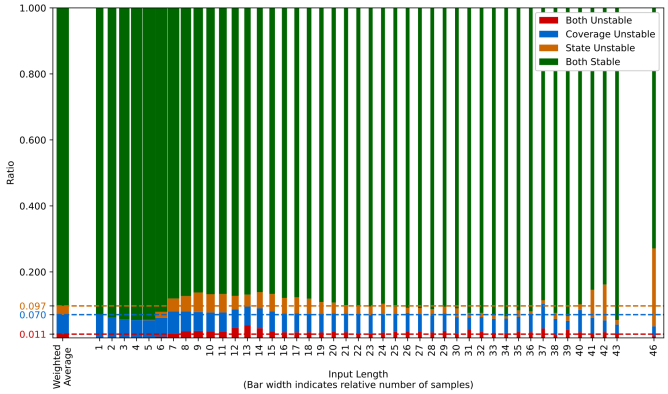

Figure 2: Average stability of both feedbacks across trace lengths

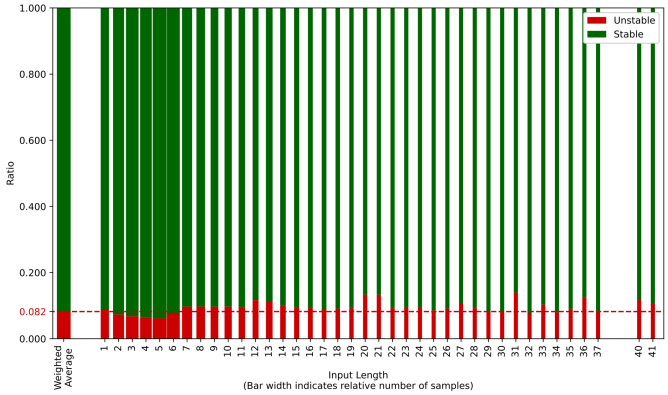

Figure 3: Average stability of coverage feedback across trace lengths

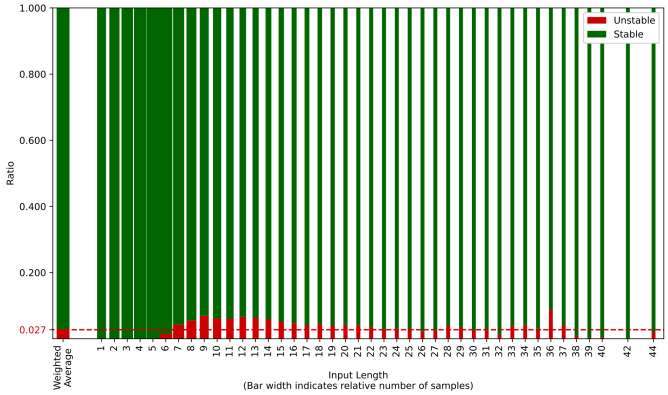

Figure 4: Average stability of state-diff feedback across trace lengths

2. `net_ipv6_mld_init` from `net/ip/ipv6_mld.c`
3. `sys_slist_init` from `sys/slist.h`
4. `z_slist_tail_set` from `sys/slist.h`
5. `net_conn_init` from `net/ip/connection.c`

Further notable entries include `net_ipv4_init`, `net_tcp_init`, and `net_init`. Across 137,571 executions of Zephyr, differences at 232,903 offsets in the coverage map were measured; the most common (`k_work_init_delayable`) appeared in 1745 executions, 73 of 386 entries found to not always be consistent appeared in fewer than 0.1% of all executions.

Discussion

Both coverage and state-diff results show inconsistencies when Zephyr is run by FTZ.

Distribution

Coverage As discussed in Section 4.2, for certain coverage entries, *how often* a certain basic block is executed is expected to be inconsistent because of inherent inconsistencies of the non-deterministic host scheduler. However, since FTZ only taints executed basic blocks, the differences measured are between a basic block being executed any number of times and the basic block not being executed at all. Functions such as `net_init` should be called on every launch, and if they are not executed, Zephyr may not be initialized correctly, thus invalidating any evaluation on it, indicating that the differences in coverage can not be ignored by FTZ. It is further notable that many of the functions in the list of inconsistently executed basic blocks appear to be related to system startup and initialization. Finally, coverage is measured with high consistency across trace length, suggesting that the logic only executed sometimes is unrelated to packet handling.

While Section 5.2 explores how different configurations of Zephyr make both state and code coverage less stable at the host's performance limit, I was unable to determine the fundamental source of these inconsistencies. Because of the high inconsistency rate in the measurements of basic block coverage, I decided not to use it as feedback to the fuzzer in FTZ. However, the values can still be used as a heuristic to measure the fuzzer's ability to test all system parts recorded at any part in any execution. Since `MaxMapFeedback` records map entries present in *any* map passed to it, even if a certain function is not called in one execution because something

No.	Time	Protocol	Source	Destination	Length	Info
1	0.000000	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
2	0.043495	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
3	0.086996	ICMPv6	::	ff02::1:…	78	Neighbor Solicitation for fe80::5eff:fe00:5331
4	0.087045	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
5	0.130551	ICMPv6	::	ff02::2	62	Router Solicitation
6	0.130570	ICMPv6	fe80::20…	::	78	Router Advertisement from 00:00:5e:00:53:ff
7	0.174075	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
8	0.217566	ICMPv6	::	ff02::1:…	78	Neighbor Solicitation for 2001:db8::1
9	0.217595	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
10	0.317841	TCP	192.0.2.2	192.0.2.1	74	41052 → 4242 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3414956140 TSecr=0 WS=128
11	0.361360	ARP	02:00:5e…	Broadcast	42	Who has 192.0.2.2? Tell 192.0.2.1
12	0.361365	ARP	ICANNIAN…	02:00:5e…	42	192.0.2.2 is at 02:00:5e:00:53:31
13	0.404856	TCP	192.0.2.1	192.0.2.2	58	4242 → 41052 [SYN, ACK] Seq=0 Ack=1 Win=1536 Len=0 MSS=1460
14	0.505074	TCP	192.0.2.2	192.0.2.1	54	41052 → 4242 [ACK] Seq=1 Ack=1 Win=64240 Len=0
15	0.605329	TCP	192.0.2.2	192.0.2.1	61	41052 → 4242 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=7
16	0.648829	TCP	192.0.2.1	192.0.2.2	54	4242 → 41052 [ACK] Seq=1 Ack=8 Win=1529 Len=0
17	0.692331	TCP	192.0.2.1	192.0.2.2	61	4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7
18	0.792593	TCP	192.0.2.2	192.0.2.1	54	24483 → 63262 [FIN, SYN, ACK, AE] Seq=0 Ack=1 Win=64236 Len=0
19	0.836089	TCP	192.0.2.1	192.0.2.2	54	63262 → 24483 [RST] Seq=1 Win=0 Len=0
20	0.938582	TCP	192.0.2.2	192.0.2.1	74	41054 → 4242 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3414956140 TSecr=0 WS=128
21	0.982101	TCP	192.0.2.1	192.0.2.2	61	[TCP Retransmission] 4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7
22	1.025987	TCP	192.0.2.1	192.0.2.2	54	4242 → 41054 [RST] Seq=1 Win=0 Len=0
23	1.102896	ICMPv6	fe80::5e…	ff02::2	70	Router Solicitation from 02:00:5e:00:53:31
24	1.102917	ICMPv6	fe80::20…	fe80::5e…	78	Router Advertisement from 00:00:5e:00:53:ff
25	1.146419	ICMPv6	fe80::5e…	ff02::1:…	86	Neighbor Solicitation for fe80::200:5eff:fe00:53ff from 02:00:5e:00:53:31
26	1.146448	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
27	1.189975	ICMPv6	fe80::5e…	ff02::1:…	86	Neighbor Solicitation for fe80::200:5eff:fe00:53ff from 02:00:5e:00:53:31
28	1.190002	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
29	1.290222	TCP	192.0.2.2	192.0.2.1	54	41052 → 4242 [ACK] Seq=1 Ack=1 Win=64240 Len=0
30	1.333727	TCP	192.0.2.1	192.0.2.2	54	4242 → 41052 [ACK] Seq=1 Ack=8 Win=1536 Len=0
31	1.410656	TCP	192.0.2.1	192.0.2.2	61	[TCP Retransmission] 4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7

No.	Time	Protocol	Source	Destination	Length	Info
1	0.000000	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
2	0.043494	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
3	0.086996	ICMPv6	::	ff02::1:…	78	Neighbor Solicitation for fe80::5eff:fe00:5331
4	0.087067	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
5	0.130567	ICMPv6	::	ff02::2	62	Router Solicitation
6	0.130588	ICMPv6	fe80::20…	::	78	Router Advertisement from 00:00:5e:00:53:ff
7	0.174104	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
8	0.218412	ICMPv6	::	ff02::1:…	78	Neighbor Solicitation for 2001:db8::1
9	0.218440	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
10	0.318665	TCP	192.0.2.2	192.0.2.1	74	41052 → 4242 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3414956140 TSecr=0 WS=128
11	0.362177	ARP	02:00:5e…	Broadcast	42	Who has 192.0.2.2? Tell 192.0.2.1
12	0.362182	ARP	ICANNIAN…	02:00:5e…	42	192.0.2.2 is at 02:00:5e:00:53:31
13	0.405706	TCP	192.0.2.1	192.0.2.2	58	4242 → 41052 [SYN, ACK] Seq=0 Ack=1 Win=1536 Len=0 MSS=1460
14	0.505963	TCP	192.0.2.2	192.0.2.1	54	41052 → 4242 [ACK] Seq=1 Ack=1 Win=64240 Len=0
15	0.606165	TCP	192.0.2.2	192.0.2.1	61	41052 → 4242 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=7
16	0.649661	TCP	192.0.2.1	192.0.2.2	54	4242 → 41052 [ACK] Seq=1 Ack=8 Win=1529 Len=0
17	0.693155	TCP	192.0.2.1	192.0.2.2	61	4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7
18	0.793417	TCP	192.0.2.2	192.0.2.1	54	24483 → 63262 [FIN, SYN, ACK, AE] Seq=0 Ack=1 Win=64236 Len=0
19	0.836926	TCP	192.0.2.1	192.0.2.2	54	63262 → 24483 [RST] Seq=1 Win=0 Len=0
20	0.937257	TCP	192.0.2.2	192.0.2.1	74	41054 → 4242 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3414956140 TSecr=0 WS=128
21	0.980767	TCP	192.0.2.1	192.0.2.2	61	[TCP Retransmission] 4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7
22	1.024283	TCP	192.0.2.1	192.0.2.2	54	4242 → 41054 [RST] Seq=1 Win=0 Len=0
23	1.124470	TCP	192.0.2.2	192.0.2.1	54	41052 → 4242 [ACK] Seq=1 Ack=1 Win=64240 Len=0
24	1.134549	ICMPv6	fe80::5e…	ff02::2	70	Router Solicitation from 02:00:5e:00:53:31
25	1.134568	ICMPv6	fe80::20…	fe80::5e…	78	Router Advertisement from 00:00:5e:00:53:ff
26	1.178064	ICMPv6	fe80::5e…	ff02::1:…	86	Neighbor Solicitation for fe80::200:5eff:fe00:53ff from 02:00:5e:00:53:31
27	1.178094	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
28	1.221590	ICMPv6	fe80::5e…	ff02::1:…	86	Neighbor Solicitation for fe80::200:5eff:fe00:53ff from 02:00:5e:00:53:31
29	1.221619	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2

No.	Time	Protocol	Source	Destination	Length	Info
1	0.000000	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
2	0.043496	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
3	0.086990	ICMPv6	::	ff02::1:…	78	Neighbor Solicitation for fe80::5eff:fe00:5331
4	0.087038	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
5	0.130527	ICMPv6	::	ff02::2	62	Router Solicitation
6	0.130539	ICMPv6	fe80::20…	::	78	Router Advertisement from 00:00:5e:00:53:ff
7	0.174036	ICMPv6	::	ff02::16	90	Multicast Listener Report Message v2
8	0.217504	ICMPv6	::	ff02::1:…	78	Neighbor Solicitation for 2001:db8::1
9	0.217527	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
10	0.317780	TCP	192.0.2.2	192.0.2.1	74	41052 → 4242 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3414956140 TSecr=0 WS=128
11	0.361269	ARP	02:00:5e…	Broadcast	42	Who has 192.0.2.2? Tell 192.0.2.1
12	0.361272	ARP	ICANNIAN…	02:00:5e…	42	192.0.2.2 is at 02:00:5e:00:53:31
13	0.406530	TCP	192.0.2.1	192.0.2.2	58	4242 → 41052 [SYN, ACK] Seq=0 Ack=1 Win=1536 Len=0 MSS=1460
14	0.506769	TCP	192.0.2.2	192.0.2.1	54	41052 → 4242 [ACK] Seq=1 Ack=1 Win=64240 Len=0
15	0.609098	TCP	192.0.2.2	192.0.2.1	61	41052 → 4242 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=7
16	0.652594	TCP	192.0.2.1	192.0.2.2	54	4242 → 41052 [ACK] Seq=1 Ack=8 Win=1529 Len=0
17	0.696031	TCP	192.0.2.1	192.0.2.2	61	4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7
18	0.796311	TCP	192.0.2.2	192.0.2.1	54	24483 → 63262 [FIN, SYN, ACK, AE] Seq=0 Ack=1 Win=64236 Len=0
19	0.839879	TCP	192.0.2.1	192.0.2.2	54	63262 → 24483 [RST] Seq=1 Win=0 Len=0
20	0.940128	TCP	192.0.2.2	192.0.2.1	74	41054 → 4242 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3414956140 TSecr=0 WS=128
21	0.983637	TCP	192.0.2.1	192.0.2.2	54	4242 → 41054 [RST] Seq=1 Win=0 Len=0
22	1.027171	TCP	192.0.2.1	192.0.2.2	61	[TCP Retransmission] 4242 → 41052 [PSH, ACK] Seq=1 Ack=8 Win=1536 Len=7
23	1.105341	ICMPv6	fe80::5e…	ff02::2	70	Router Solicitation from 02:00:5e:00:53:31
24	1.105361	ICMPv6	fe80::20…	fe80::5e…	78	Router Advertisement from 00:00:5e:00:53:ff
25	1.148859	ICMPv6	fe80::5e…	ff02::1:…	86	Neighbor Solicitation for fe80::200:5eff:fe00:53ff from 02:00:5e:00:53:31
26	1.148887	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
27	1.192476	ICMPv6	fe80::5e…	ff02::1:…	86	Neighbor Solicitation for fe80::200:5eff:fe00:53ff from 02:00:5e:00:53:31
28	1.192508	ICMPv6	fe80::20…	ff02::16	110	Multicast Listener Report Message v2
29	1.292761	TCP	192.0.2.2	192.0.2.1	54	41052 → 4242 [ACK] Seq=1 Ack=1 Win=64240 Len=0
30	1.336266	TCP	192.0.2.1	192.0.2.2	54	4242 → 41052 [ACK] Seq=1 Ack=8 Win=1536 Len=0

Figure 5: Packets exchanged for the same input in 10 executions, recorded 4, 4, and 2 times respectively

went wrong, for example, during Zephyr's initialization logic, the same function is likely to be triggered in one of the other executions.

State-Diff The differences, as measured by the state-diff observer, indicate a difference in packets received by the fuzzer. This could be confirmed by manual inspection of the different interactions between the fuzzer and Zephyr from the same input. As described in Sections 4.6.2 and 4.6.3, FTZ provides functionality to extract these into `pcap` files. Figure 5 shows screenshots of three exchanges recorded for 10 executions with the same input.

Like in the example, retransmitted packets and those resetting the connection appear frequently. However, I was not able to find a reason for the inconsistencies or a definite pattern between them.

Unlike basic block coverage, state-diff coverage seems trace-length dependent: Traces with length ≤ 5 are almost always consistent with respect to the packets exchanged between fuzzer and Zephyr. However, the ratio does not continue rising with increased trace length after that, as would be expected if the error rate is independent of state and packet. This suggests that an unknown factor remains influencing the reliability of Zephyr's responses. The fact that short traces are executed comparatively consistently may be caused by either a filling buffer in Zephyr or an underlying problem requiring Zephyr to be in a state only reachable with a certain number of packets.

Distribution The consistencies of both feedbacks show a further notable statistic: There seems to be a set of inputs triggering inconsistent behavior more likely than others. Figure 6 shows the distribution of ratios between the incorrect and correct measurements across trace lengths. For an input that provoked three different observer values appearing 2, 10, and 100 times, this ratio would therefore be 0.12. Two sections along the y-axis appear with higher densities, at approximately 0.1 and 0.75 respectively, with fewer inputs resulting in consistencies at around 0.4. The same effect does not appear when coverage feedback is used (see Figure 8) and is significantly less pronounced when using both feedbacks (see Figure 7). The latter can be explained by coverage being less consistent than state-diff, overpowering its effects.

I was unable to create sufficient proof of this conjecture during this thesis, nor find a correlation between the actual traces that would explain this behavior.

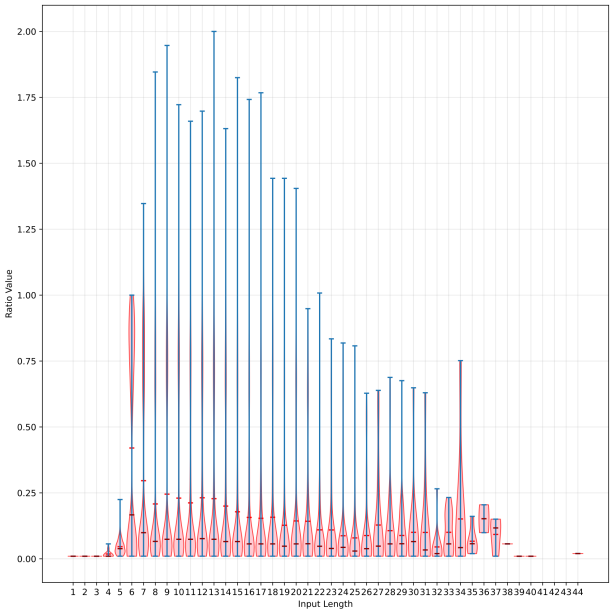

Figure 6: Ratio between incorrect and correct observer measurements across trace lengths when using state-diff feedback (violin width indicates density)

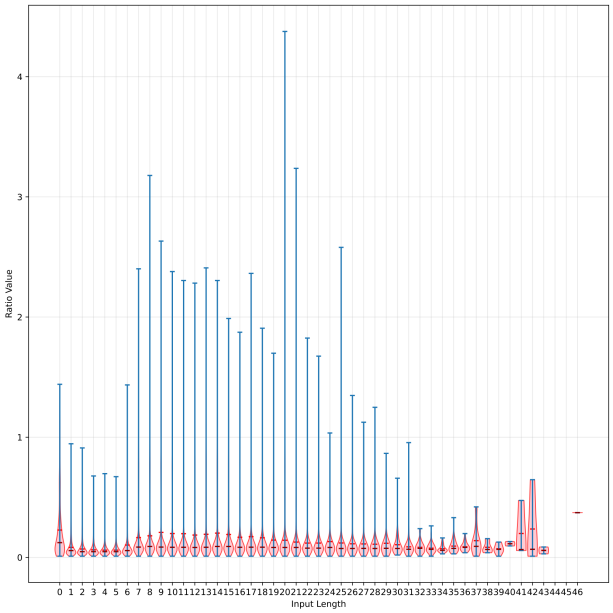

Figure 7: Ratio between incorrect and correct observer measurements across trace lengths when using both feedbacks (violin width indicates density)

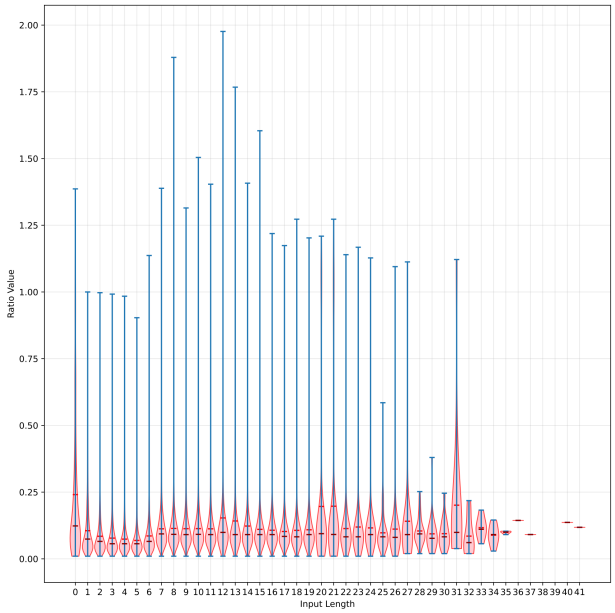

Figure 8: Ratio between incorrect and correct observer measurements across trace lengths when using coverage feedback (violin width indicates density)

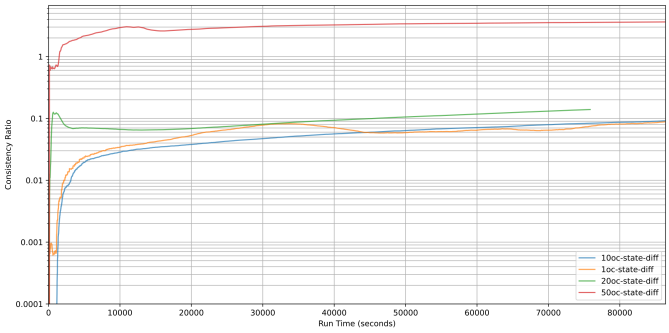

Figure 9: Consistencies across different overcommit values

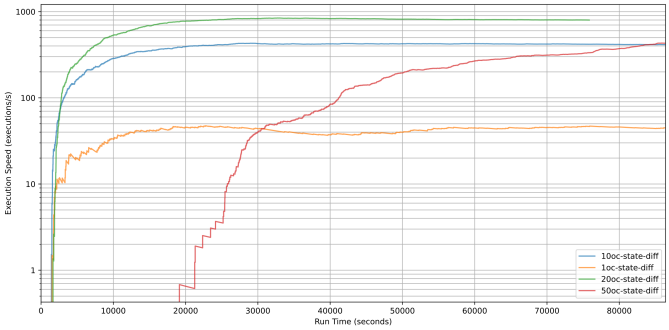

Figure 10: Execution speed across different overcommit values

5.2 Throughput and Overcommit

When spawning FTZ with a single fuzzing client per core, as usual in fuzzing, the CPU time load averages approximately 0.8% in user space and 1.9% in system space. Increasing the `overcommit` value and thus the number of clients per core increases instability rates for both observers, as seen in Figure 9. However, it simultaneously increases the overall execution speed of the fuzzer up to a certain point, as seen in Figure 10. The CPU loads for `overcommit` values of 10, 20, and 50 are 3.9%/10.1%, 8.7%/23.1%, and 8.3%/89.9%, respectively.

Discussion

As described in Section 4.6.2, executing a single input on Zephyr leads to a dominating number of wait states in both Zephyr and the fuzzer needed for synchronization. The inconsistency rate even increases when the number of clients per core is only increased to a level where the CPU is far from fully loaded. This suggests the limitation of FTZ is in scheduling the number of processes requiring precise timing for their execution consistency. When fully loading the CPU, the execution speed collapses by approximately a factor of 10 compared to linear scaling, suggesting a bottleneck besides CPU load.

To mitigate increased inconsistency rates, `ReplayingFuzzer` has to be configured to re-execute each input more often, slowing throughput of new inputs and thus negating some (or even all) of the gained speed. Since the error rates remain approximately consistent for `overcommit` values of up to 10, the remaining evaluations were done with FTZ configured to run 10 clients per core.

5.3 State Feedback vs. State-Diff Feedback

Section 4.5 presents two ways of using the information extracted from incoming packets: either directly or by inferring state transitions. Figure 11 shows the coverage achieved by the different configurations of FTZ. The most prominent effect of changing a single configuration is between choosing direct state or state-diff feedback. Note that FTZ is configured to measure coverage across all parts of Zephyr and is therefore unlikely to achieve code coverage of 100%, as many parts of the logic are not currently examined by FTZ.

Figure 12 shows that FTZ cannot find any further improvements in direct states after a few hours. Figure 14 shows a similar pattern, where no inputs are added to the corpus after the same time cutoff. This

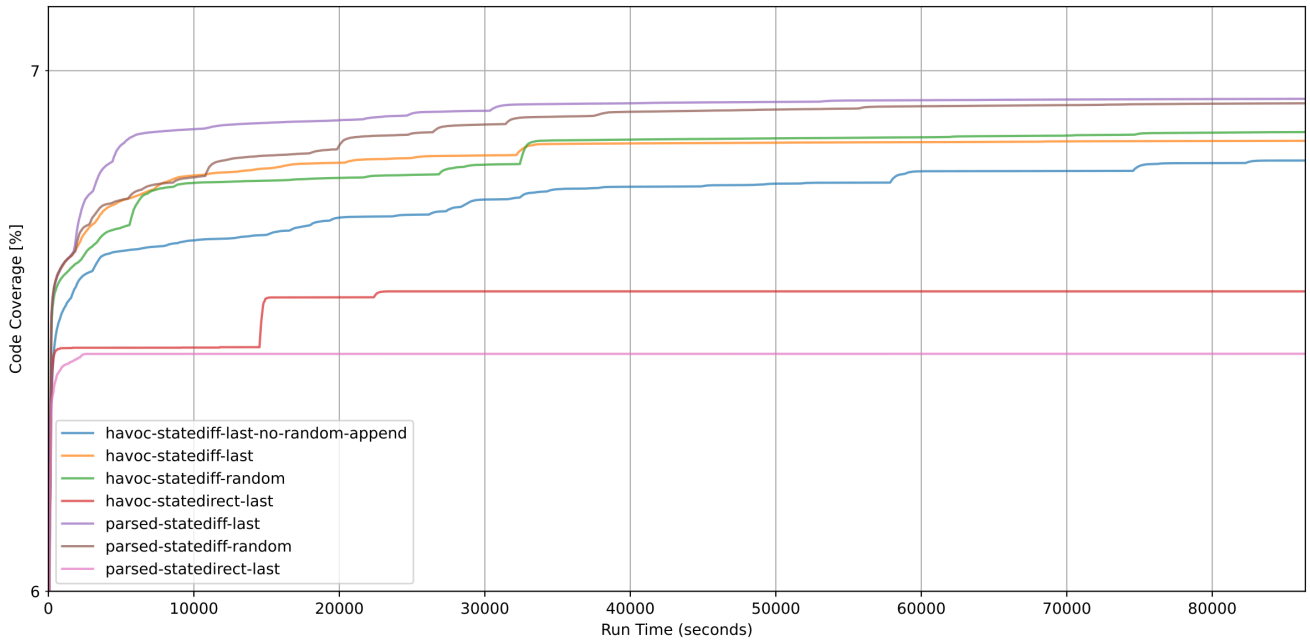

Figure 11: Code coverage across different configurations

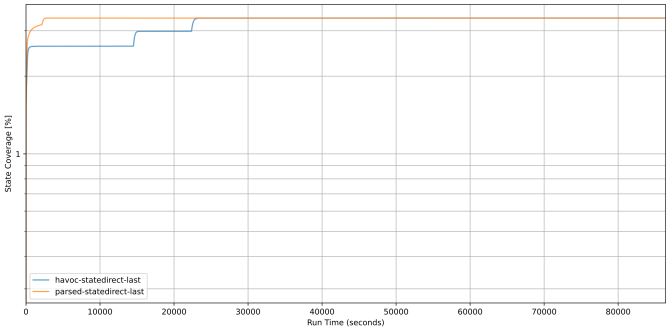

Figure 12: States in replies recorded by FTZ when configured using direct state feedback

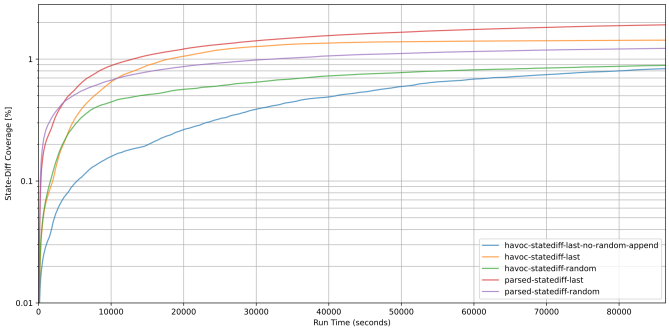

Figure 13: State-diff feedback covered by different configurations of FTZ

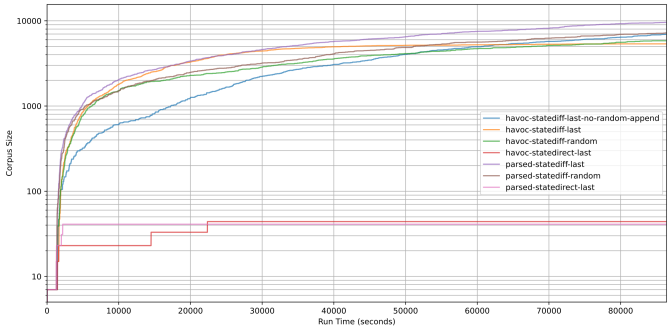

Figure 14: Number of corpus entries across different configurations

cannot be said for state-diff coverage, as seen in Figure 13.

Discussion

The number of states that Zephyr can emit seems too small to make direct state feedback an effective feedback mechanism for FTZ. Note that FTZ is unlikely to achieve either a state or state-diff coverage of 100%, as Zephyr is not built to emit every combination of flags in the TCP header. All achievable states are exhausted within a few hours, including propagation of the found inputs to all 640 clients. Even still, simple direct state feedback may improve a fuzzer if it is combined with another form of feedback, like code

coverage. However, this was not further evaluated due to the unreliability of coverage measurements in FTZ (see Section 5.1).

State-diff feedback performs significantly better at guiding FTZ to previously untested program parts. It both reaches significantly better code coverage, and while its behavior shows a slowing of discovering inputs that trigger additional state-diff values, within 24 hours, it does not reach a plateau similar to using direct state feedback.

5.4 Mutation Target Selection

Section 4.4.1 introduced how FTZ can be configured to mutate either a random message in a trace or always focus on the last message. Figure 11 shows a slight difference in performance between the two options for both byte array and parsed message representation. While the difference is more prominent in the early stages of the fuzzing campaign, selecting the last message as a mutation target is shown to be more effective even after 24 hours. Figure 13 further demonstrates that the strategy of always mutating the last message is also more effective and efficient at covering all possible state-diff map entries.

Discussion

When randomly selecting a message to mutate, FTZ with a certain probability selects the last message, which may explain why, after a while, it catches up to the alternative strategy of always mutating the last message. This further suggests that the assumption described in Section 4.4.1 holds, and mutating early messages in long traces often renders the remainder of the trace useless, as the interdependencies between states and messages are broken.

5.5 Input Modeling and Mutation

Recall that individual messages can be modeled in two ways, as discussed in Section 4.4.3: binary data or a parsed structure containing individual fields of the headers of the different layers. The choice of input modeling further influences the applicable mutators. Binary data is mutated randomly, while FTZ retains the basic structure of packets when mutating on parsed input representations. Either set of mutators is extended by two generic mutators that both append messages to a trace — either copied from the seed recording or with randomly generated yet valid data.

Figures 11 and 13 show that modeling messages as a parsed data structure is more effective and efficient at maximizing code coverage and state-diff coverage. When FTZ is configured to use binary message representations, and the mutator appending randomly generated messages is disabled, performance decreases significantly compared to a run with both appending mutators enabled.

Discussion

When modeling messages as a parsed structure, FTZ is severely limited in the possible input space it can explore. It can only output fixed layer 2 and 3 headers (except for their fields that depend on the packet contents), and even the TCP header always has a correct length field and checksum. However, these experiments show that this helps FTZ to test more parts of Zephyr.

Byte array modeling of inputs and random mutation is more powerful in theory — it can generate inputs that invalidate only one of the fixed values listed above — but it is very unlikely to generate the correct values for all others randomly. This further means that Zephyr will reject most packets generated by such mutation in its first parsing layer while deeper logic is never reached.

The starkly diminished results from the experiment where FTZ is configured without the mutator appending generated messages to a trace can be explained by the following: Imagine two traces in the corpus, one a copy of the other, with an additional valid message at the end. If one only considers the TCP flags, appending a valid message to the shorter trace with random flags is equivalent to randomizing the flags of the longer — the latter of which is only possible with structured mutation, as it requires recomputing the checksum to remain valid. This further increases the performance difference in effectiveness between pure structure-unaware mutation and structure-aware mutation.

6 Conclusion

Section 1.1 introduces a set of research questions, which have been explored throughout this thesis:

RQ 1a, 1b, and 1c Section 4.1 introduces `native_sim`, a wrapper that allows running Zephyr natively. In conjunction with this, FTZ uses a custom ethernet driver based on shared memory to exchange packets with the fuzzer, as introduced in Section 4.3. Implementation details are given in Section 4.6. Sections 4.1 and 4.2 show how Zephyr is instrumented with AddressSanitizer to capture additional memory errors and with CoverageSanitizer to provide coverage feedback to the fuzzer.

However, Zephyr does not behave consistently when executed by FTZ. Sections 5.1 and 5.2 present initial research into the extent and source of these inconsistencies, along with additional performance testing.

RQ 2a and 2b Section 4.5 presents how FTZ uses incoming packets in a heuristic to provide state feedback to the fuzzer. Section 5.3 evaluates this heuristic, using its results both directly and to infer state transitions, with the latter showing improved performance.

RQ 3 Section 2.2.2 introduces the structure of a TCP/IP packet, including the dependencies between fields of the headers and other fields and other headers such as checksums and length fields, and even other packets such as sequence numbering. Based on this insight and the general inability of random mutation on binary data to generate valid values for these fields, Section 4.4.3 shows two ways for FTZ to model and mutate messages: with binary data and random mutation and with parsed structures ensuring the validity of the values of certain fields and structure-aware mutation. Section 5.5 presents how the latter is more efficient and effective in reaching code and state(-diff) coverage.

RQ 4 Related works testing TCP/IP stacks propose two strategies for selecting which message to mutate in a trace. FTZ supports both selecting a random message and always selecting the last one, as presented in Section 4.4.1. Section 5.4 finally presents how the latter strategy reaches higher code and state-diff feedback faster.

6.1 Limitations and Future Work

While this thesis presents a functional proof-of-concept fuzzer in FTZ, there remain a set of improvements to implement and questions it was unable to answer.

First, several performance optimizations could be implemented in FTZ, such as replacing sleep-interrupted busy waiting in both the fuzzer and Zephyr with a form of kernel-supplied synchronization such as semaphores or incremental packet parsing. These were not completed because FTZ is not primarily constrained by CPU load but instead consistency.

6.1.1 Improved Seeding

Prior work has shown that the seeds used in a fuzzer significantly impact the fuzzer's performance [74]. FTZ currently relies on subsets of a single recorded interaction. These could be extended with other traces, specifically those exercising different paths through the state machine. Canceled or otherwise incomplete interactions, for example, may benefit the performance of FTZ.

Additional seeding could further improve the range of packets supported by the fuzzer: Support for TCP over IPv6 would require only little additional engineering in combination with additional seeds. Similarly, options in the different protocol headers (see Section 2.2.2) are currently only partially fuzzed when using parsed input structures, which additional seeds could improve in addition to a list of additional mutators.

6.1.2 Additional Targets and Comparison to Other Fuzzers

TCP-Fuzz [56] correctly identified that TCP/IP stacks have an input space larger than that of the packets they process in their configuration and invocation through system calls (see Section 3.3). These are currently not mutated in FTZ but instead fixed to the configuration of the `sockets/echo` sample (see Section 4).

FTZ further could be used as a basis to fuzz other protocols at different networking layers, such as application layer protocol implementations of an FTP or SMTP server or additional transport layer protocols like QUIC. This may allow comparison between FTZ and previous work on these protocols (refer to Section 3.2).

However, the same challenges outlined in Section 3.2.3 remain. Comparing FTZ to other approaches would either require reengineering other fuzzers' target interactions to be compatible with Zephyr or extracting the improvements made by FTZ and building a fuzzer around them targeting other code. Improvements made to LibAFL during this project make protocol fuzzing easier, but it still requires setting up the appropriate parsed data structures and mutators. However, this work could further confirm the results described in Section 5 independent of Zephyr or TCP/IP.

Alternatively, FTZ could be evaluated on other implementations of TCP/IP stacks, such as userland implementations (Section 4.6.3 discusses how one such implementation is already partially integrated into FTZ), or the network stacks of other OSs. The main challenge in this is the adaptation of the target similar to what is described in Section 4, such as sanitizer and code coverage instrumentation and drivers for the shared memory-based pseudo-networking or an alter-

nate way of exchanging packets. Access to alternate implementations would also allow FTZ to implement differential fuzzing, testing the behavior of the different targets against each other to find logic errors that do not corrupt the memory and thus cannot currently be found by FTZ.

6.1.3 Checks on and Alternate Uses for State Feedback

The effectiveness of the state-inferring heuristic presented in this thesis could be evaluated by introducing direct state feedback similar to Ijon and other related works (refer to Section 3.2.1) and comparing their effects on the fuzzer's performance. Finally, one could investigate how state information could be used besides feedback. Section 3.2.2 discusses advancements other projects have shown by using state information in other parts of the fuzzer, such as the scheduler.

6.2 Contributions and Summary

This thesis presents FTZ, a proof-of-concept fuzzer targeting the TCP/IP stack of Zephyr. To achieve this, it introduces a custom ethernet driver based on shared memory to reduce host kernel dependencies. `native_sim`, the environment simulation wrapper, is used to run Zephyr without the need for a dynamic emulation layer. It is built on and extends the fuzzing framework LibAFL.

In addition to the initial investigation into the reasons for Zephyr's at times unpredictable behavior, I evaluated the differences in performance of mutations on serialized and parsed packet representations, two strategies to select which message in a trace to mutate, and two heuristics to estimate the TCP state of a connection based on returned packets.

The experiments show that even limited structure-aware mutation is more efficient at covering both code coverage and the state space than naïve random mutation. In addition to appending new packets, mutating the last message of a trace is more efficient than selecting a random message to mutate. Finally, I present that maximizing simple state feedback as calculated by the heuristic quickly saturates the possible state space, after which FTZ no longer makes progress. Conversely, maximizing coverage across state transitions proves to be more effective.

In the interest of open science, the source code and all artifacts produced during this project are publicly available and released under an open-source license. During development, thousands of lines of code have been introduced to multiple upstream projects. All artifacts produced for this project are available at

github.com/riesentoaster/fuzzing-zephyr-network-stack.**1.change url 2.Re-render plots at higher DPI, fix placement**

Bibliography

[1] "About the zephyr project." (n.d.), [Online]. Available: https://www.zephyrproject.org/learn-about/ (visited on Jan. 30, 2025).

[2] "Real-time operating system." (n.d.), [Online]. Available: https://en.wikipedia.org/wiki/Real-time_operating_system (visited on Jan. 30, 2025).

[3] "Products running zephyr." (n.d.), [Online]. Available: https://www.zephyrproject.org/products-running-zephyr/ (visited on Jan. 30, 2025).

[4] T. Scharnowski, N. Bars, M. Schloegel, *et al.*, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.

[5] Y. Shen, Y. Xu, H. Sun, *et al.*, "Tardis: Coverage-guided embedded operating system fuzzing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4563–4574, 2022. DOI: 10.1109/TCAD.2022.3198910.

[6] P. C. Amusuo, R. A. C. Méndez, Z. Xu, A. Machiry, and J. C. Davis, "Systematically detecting packet validation vulnerabilities in embedded network stacks," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 926–938. DOI: 10.1109/ASE56229.2023.00095.

[7] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, Oct. 2023, ISSN: 0360-0300. DOI: 10.1145/3623375. [Online]. Available: https://doi.org/10.1145/3623375.

[8] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," Nov. 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[9] R. McNally, K. K.-H. Yiu, D. A. Grove, and D. Gerhardy, "Fuzzing: The state of the art," *DSTO Defence Science and Technology Organisation*, Feb. 2012.

[10] J. Methman. "Clusterfuzzlite: Continuous fuzzing for all." (Nov. 2021), [Online]. Available: https://security.googleblog.com/2021/11/clusterfuzzlite-continuous-fuzzing-for.html (visited on Feb. 5, 2025).

[11] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[12] "Google scholar — fuzz testing." (2025), [Online]. Available: https://scholar.google.com/scholar?q=fuzz+testing (visited on Jan. 30, 2025).

[13] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[14] "Syzkaller - kernel fuzzer." (n.d.), [Online]. Available: https://github.com/google/syzkaller (visited on Feb. 5, 2025).

[15] "Libfuzzer — a library for coverage-guided fuzz testing." (n.d.), [Online]. Available: https://llvm.org/docs/LibFuzzer.html (visited on Feb. 5, 2025).

[16] "Honggfuzz." (n.d.), [Online]. Available: https://honggfuzz.dev (visited on Feb. 5, 2025).

[17] "Oss-fuzz." (n.d.), [Online]. Available: https://google.github.io/oss-fuzz/ (visited on Feb. 5, 2025).

[18] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024, Retrieved 2024-07-01 16:50:18+02:00. [Online]. Available: https://www.fuzzingbook.org/.

[19] P. Cai, "Kernel- vs. user-level networking: A ballad of interrupts and how to mitigate them," M.S. thesis, University of Waterloo, 2023.

[20] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018. DOI: 10.1109/TR.2018.2834476.

[21] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[22] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, vol. 19, 2019, pp. 1–15.

[23] C. Daniele, S. B. Andarzian, and E. Poll, "Fuzzers for stateful systems: Survey and research directions," *ACM Comput. Surv.*, vol. 56, no. 9, Apr. 2024, ISSN: 0360-0300. DOI: 10.1145/3648468. [Online]. Available: https://doi.org/10.1145/3648468.

[24] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021. DOI: 10.1109/MS.2020.3016773.

[25] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3255–3272, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[26] B. Zhao, Z. Li, S. Qin, *et al.*, "StateFuzz: System Call-Based State-Aware linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3273–3289, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong.

[27] C. Lyu, S. Ji, C. Zhang, *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[28] "American fuzzy lop — whitepaper." (n.d.), [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on Feb. 5, 2025).

[29] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22, Los Angeles, U.S.A.: ACM, Nov. 2022.

[30] W. Zhou, S. Shen, and P. Liu, "Iot firmware emulation and its security application in fuzzing: A critical revisit," *Future Internet*, vol. 17, no. 1, Jan. 2025. DOI: 10.3390/fi17010019.

[31] J. Kim, J. Yu, Y. Lee, D. D. Kim, and J. Yun, "Hd-fuzz: Hardware dependency-aware firmware fuzzing via hybrid mmio modeling," *Journal of Network and Computer Applications*, vol. 224, p. 103 835, 2024, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2024.103835. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804524000122.

[32] G. Farrelly, M. Chesser, and D. C. Ranasinghe, "Ember-io: Effective firmware fuzzing with model-free memory mapped io," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23, Melbourne, VIC, Australia: Association for Computing Machinery, 2023, pp. 401–414, ISBN: 9798400700989. DOI: 10.1145/3579856.3582840. [Online]. Available: https://doi.org/10.1145/3579856.3582840.

[33] G. Farrelly, P. Quirk, S. S. Kanhere, S. Camtepe, and D. C. Ranasinghe, "Splits: Split input-to-state mapping for effective firmware fuzzing," in *Computer Security – ESORICS 2023*, G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis, Eds., Cham: Springer Nature Switzerland, 2024, pp. 290–310, ISBN: 978-3-031-51482-1.

[34] Z. Chen, "Security of esoteric firmware and trusted execution environments," Ph.D. dissertation, University of Birmingham, http://etheses.bham.ac.uk/id/eprint/13842, 2023.

[35] M. Chesser, S. Nepal, and D. C. Ranasinghe, "MultiFuzz: A Multi-Stream fuzzer for testing monolithic firmware," in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5359–5376, ISBN: 978-1-939133-44-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/chesser.

[36] "Native simulator — repository." (n.d.), [Online]. Available: https://github.com/BabbleSim/native_simulator/ (visited on Jan. 31, 2025).

[37] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.

[38] V. Herdt, D. Große, J. Wloka, T. Güneysu, and R. Drechsler, "Verification of embedded binaries using coverage-guided fuzzing with systemc-based virtual prototypes," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20, Virtual Event, China: Association for Computing Machinery, 2020, pp. 101–106, ISBN: 9781450379441. DOI: 10.1145/3386263.3406899. [Online]. Available: https://doi.org/10.1145/3386263.3406899.

[39] V. Herdt and R. Drechsler, "Efficient techniques to strongly enhance the virtual prototype based design flow," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 182–187. DOI: 10.1109/ISVLSI49217.2020.00041.

[40] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on iot device firmware: A survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 1, pp. 25–41, 2023. DOI: 10.1109/JAS.2022.105860.

[41] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, Jan. 2021, ISSN: 0360-0300. DOI: 10.1145/3423167. [Online]. Available: https://doi.org/10.1145/3423167.

[42] A. Fasano, T. Ballo, M. Muench, *et al.*, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 687–701, ISBN: 9781450382878. DOI: 10.1145/3433210.3453093. [Online]. Available: https://doi.org/10.1145/3433210.3453093.

[43] W.-L. Huang and K. G. Shin, *Es-fuzz: Improving the coverage of firmware fuzzing with stateful and adaptable mmio models*, 2024. arXiv: 2403.06281 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2403.06281.

[44] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "Hoedur: Embedded firmware fuzzing using multi-stream inputs," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23, Anaheim, CA, USA: USENIX Association, 2023, ISBN: 978-1-939133-37-3.

[45] "Zephyr documentation — fuzzing." (n.d.), [Online]. Available: https://docs.zephyrproject.org/latest/samples/subsys/debug/fuzz/README.html (visited on Feb. 6, 2025).

[46] "Zephyr sample project — fuzz." (n.d.), [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/subsys/debug/fuzz (visited on Feb. 6, 2025).

[47] "Fuzzbuzz — zephyr's fuzzing example." (n.d.), [Online]. Available: https://github.com/fuzzbuzz/fuzz-zephyr (visited on Feb. 6, 2025).

[48] "Fuzzing zephyr with afl and renode." (Oct. 2023), [Online]. Available: https://www.zephyrproject.org/fuzzing-zephyr-with-afl-and-renode/ (visited on Feb. 6, 2025).

[49] X. Zhang, C. Zhang, X. Li, *et al.*, "A survey of protocol fuzzing," *ACM Comput. Surv.*, vol. 57, no. 2, Oct. 2024, ISSN: 0360-0300. DOI: 10.1145/3696788. [Online]. Available: https://doi.org/10.1145/3696788.

[50] X. Wei, Z. Yan, and X. Liang, "A survey on fuzz testing technologies for industrial control protocols," *Journal of Network and Computer Applications*, vol. 232, p. 104 020, 2024, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2024.104020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804524001978.

[51] R. Natella and V.-T. Pham, "ProFuzzBench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 662–665, ISBN: 9781450384599. DOI: 10.1145/3460319.3469077. [Online]. Available: https://doi.org/10.1145/3460319.3469077.

[52] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, Oct. 2022, ISSN: 1573-7616. DOI: 10.1007/s10664-022-10233-3. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3.

[53] R. Helmke, E. Winter, and M. Rademacher, "Epf: An evolutionary, protocol-aware, and coverage-guided network fuzzing framework," in *2021 18th International Conference on Privacy, Security and Trust (PST)*, 2021, pp. 1–7. DOI: 10.1109/PST52912.2021.9647801.

[54] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[55] V. Paliath, E. Trickel, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "Sandpuppy: Deep-state fuzzing guided by automatic detection of state-representative variables," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, F. Maggi, M. Egele, M. Payer, and M. Carminati, Eds., Cham: Springer Nature Switzerland, 2024, pp. 227–250, ISBN: 978-3-031-64171-8.

[56] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 489–502, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou.

[57] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1024–1036, ISBN: 9781450371216. DOI: 10.1145/3377811.3380421. [Online]. Available: https://doi.org/10.1145/3377811.3380421.

[58] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465. DOI: 10.1109/ICST46399.2020.00062.

[59] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with data dependency information," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022, pp. 286–302. DOI: 10.1109/EuroSP53844.2022.00026.

[60] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in tcp congestion control using a model-guided approach," in *Proceedings of the 2018 Applied Networking Research Workshop*, ser. ANRW '18, Montreal, QC, Canada: Association for Computing Machinery, 2018, p. 95, ISBN: 9781450355858. DOI: 10.1145/3232755.3232769. [Online]. Available: https://doi.org/10.1145/3232755.3232769.

[61] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," in *2008 IEEE International Conference on Network Protocols*, 2008, pp. 114–123. DOI: 10.1109/ICNP.2008.4697030.

[62] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *Ijcsns*, vol. 10, no. 8, p. 239, 2010.

[63] P. Zhao, S. Jiang, S. Liu, and L. Liu, "Fuzz testing of protocols based on protocol process state machines," in *2024 4th International Conference on Electronic Information Engineering and Computer Science (EIECS)*, 2024, pp. 844–850. DOI: 10.1109/EIECS63941.2024.10800590.

[64] D. Maier, O. Bittner, J. Beier, and M. Munier, "Fitm: Binary-only coverage-guided fuzzing for stateful network protocols," Workshop on Binary Analysis Research, Internet Society, 2022. DOI: 10.14722/bar.2022.23008. [Online]. Available: http://dx.doi.org/10.14722/bar.2022.23008.

[65] A. G. Voyiatzis, K. Katsigiannis, and S. Koubias, "A modbus/tcp fuzzer for testing internetworked industrial systems," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2015, pp. 1–6. DOI: 10.1109/ETFA.2015.7301400.

[66] Y. Lai, H. Gao, and J. Liu, "Vulnerability mining method for the modbus tcp using an anti-sample fuzzer," *Sensors*, vol. 20, no. 7, 2020, ISSN: 1424-8220. DOI: 10.3390/s20072040. [Online]. Available: https://www.mdpi.com/1424-8220/20/7/2040.

[67] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: A gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18, Ischia, Italy: Association for Computing Machinery, 2018, pp. 138–145, ISBN: 9781450357616. DOI: 10.1145/3203217.3203241. [Online]. Available: https://doi.org/10.1145/3203217.3203241.

[68] W. Wang, Z. Chen, Z. Zheng, H. Wang, and J. Luo, "MTA Fuzzer: A low-repetition rate modbus tcp fuzzing method based on transformer and mutation target adaptation," *Computers & Security*, vol. 144, p. 103 973, 2024, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2024.103973. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824002785.

[69] K. Katsigiannis and D. Serpanos, "Mtf-storm: A high performance fuzzer for modbus/tcp," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 926–931. DOI: 10.1109/ETFA.2018.8502600.

[70] Q. Xiong, H. Liu, Y. Xu, *et al.*, "A vulnerability detecting method for modbus-tcp based on smart fuzzing mechanism," in *2015 IEEE International Conference on Electro/Information Technology (EIT)*, 2015, pp. 404–409. DOI: 10.1109/EIT.2015.7293376.

[71] "Native simulator - native_sim." (n.d.), [Online]. Available: https://docs.zephyrproject.org/latest/boards/native/native_sim/doc/index.html (visited on Jan. 31, 2025).

[72] "Clang documentation — addresssanitizer." (n.d.), [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html (visited on Jan. 31, 2025).

[73] "Zephyr — repository — issue: Runtime failure on samples/net/sockets/echo with llvm 16 and asan for native_sim." (n.d.), [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/issues/83863 (visited on Jan. 31, 2025).

[74] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 230–243, ISBN: 9781450384599. DOI: 10.1145/3460319.3464795. [Online]. Available: https://doi.org/10.1145/3460319.3464795.