FTZ: A Stateful Fuzzer for the TCP/IP Stack of Zephyr

Valentin Huber

at Cyber Defence Campus

and Institute of Computer Science at ZHAW

contact@valentinhuber.me **1.fix title page**

February 10, 2025

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.**2.write**

Keywords: Software Testing, Fuzzing, 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 programers to define time constraints on their code, which are then guaranteed by the process scheduler. To do this, processes have to be verifiably complete within a constant maximal time or fail safely. This is particularly important for safety-critical systems where delays in computation can pose serious risk, such as control of a robot. [2] Zephyr is used in a wide range of products, including large industrial equipments such as wind turbines, medtech devices such as hearing aids, laptops, or embedded devices such as emergency water detectors. [3]

Zephyr projects typically connect to other devices or the internet using one of the OS's networking libraries, such as a custom-built TCP/IP stack. Since many of these devices need access to the internet, any defect in the network stack is comparatively likely to be remotely exploitable and thus critical. In combination with the potentially dangerous devices Zephyr runs on, this makes the correctness of these parts of Zephyr paramount.**3.Too much opinion?**

Fuzzing has been established as a widely used technique [4] to find software defects by repeatedly running a certain 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: Windows requires dominant software to be fuzzed prior to release [4], and its fuzzer SAGE [5] "reportedly found a third of the Windows 7 bugs between 2007-2009" [6]. Google admitted to finding 20,000 vulnerabilities in Chrome alone using fuzz testing [4], 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 critical projects [7].

Ever since its inception in the nominal work by Miller *et al.* [8], 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" [9]) include observing the program execution to guide the fuzzer when selecting the next input to test, or improved oracles such as memory sanitizers. Section 2.1 provides additional background on 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.

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 advanced input representation and mutation techniques, stateful feedback, and reliability-enhancing logic to effectively test Zephyr. The remainder of this thesis is structured as follows:

- Section 2 presents further background on fuzzing in general and targeting TCP/IP systems in particular,
- Section 3 explores related works,
- Section 4 explains the different techniques used in FTZ, along with how they were implemented,
- Section 5 presents the results achieved in the evaluation of FTZ,
- Section 6 discusses the implication of those results on the effectiveness of FTZ, and
- Section 7 summarizes the findings of this thesis.

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 use across a wide range of influential fuzzers, such as AFL++ [10], syskaller [11], libFuzzer [12], Honggfuzz [13], or OSS-Fuzz [14]. 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 in some way, and evaluates it in the PUT.

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 mixture between blackbox fuzzers, which do not know anything about the internals of the PUT during the execution, and whitebox fuzzers, which further analyze the executed logic using symbolic execution or taint analysis. [15]**4.Can I just cite my own work here?**

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 was measured in the PUT. This incentivizes the fuzzer to explore previously untested parts of the PUT. [10]

2.2 Fuzzing Network Stacks Is Hard

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

2.2.1 Deep integration with OS

OS' built-in network stacks are deeply integrated with other subsystems, such as the memory management subsystem for accessing the network interface card and socket buffers, the system call-based API thought the virtual filesystem. It further relies on the system's scheduler for event notifications. Extracting the network stack from the rest of the kernel to be run independently is therefore infeasible, since it would require extensive additional engineering, which is error-prone and may result in findings that do not hold on the original implementation. [16]

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 sanitizer 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.

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 with. Many of the fields across all layers depend on values of other fields in their own or even other layers. Examples of these 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 a prime example of logic fuzzers struggle to penetrate. This is because mutating one part of the data requires very specific changes to another, which is highly unlikely to happen in random mutation. [17] This challenge has garnered some attention from previous work, such as removing the program sections responsible for checking checksums from the target program [18], or bypassing them [19].

The sequence and acknowledgment numbers in the TCP header are dependent not only on the current packet, but 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. [20] Indeed, stateful targets are listed among the remaining challenges in fuzzing by Boehme *et al.* in their review paper. [21] Ba *et al.* demonstrate that simply relying on coverage information is insufficient to fully test stateful code, but the state of the target has to be taken into account. [22] A similar conclusion was drawn during the evaluation of StateFuzz. [23]

TCP being a stateful protocol introduces two fundamental challenges to the architecture of the fuzzer:

1. To retain the ability to triage the software error found by the fuzzer, one needs the ability to repeatedly cause the same error. Since the PUT may be put into different states with each message, this means 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 trace evaluation, trading off the additional runtime for increased simplicity and reliability, and reduced memory and storage consumption.

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

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 network package as used during seeding of FTZ. Gray fields are optional, green fields length-dependent, and blue fields checksums.

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" [20]. 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." [20]

2.3 LibAFL

LibAFL is a fuzzing library written in Rust by the maintainers of 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, their 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 default basic default implementation of each. Additionally, many advanced algorithms such as the mutation scheduler from MOpt [24], the weighted input scheduler from AFL++ [10], Hitcount coverage postprocessing from AFL [25], or execution-dependent mutation from REDQUEEN [19] have been added. InNov. 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. Section 4 presents the details of how LibAFL is used in FTZ. [26]

3 Related Works

3.1 Fuzzing Zephyr

I was unable to find scientific work evaluating fuzz testing on Zephyr. There are however reports of fuzzing campaigns available, listed and compared to FTZ here:

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. [27], [28]

There is also an adaption of this example that allows using FuzzBuzz as the fuzzer backing the execution instead of libFuzzer. [29]

According to [30], the libFuzzer integration has successfully been used to test Zephyr's bluetooth stack. While the blog post mentioning the campaign claims that it helped finding several bugs, it does not provide any detail about the campaign, employed techniques, or evaluation. Besides targeting a different part of Zephyr, it uses libFuzzer, which is a pure coverage-guided fuzzer without any state feedback, so this campaign likely did not use state feedback.

The blog post further describes using the Renode hardware systems simulator in combination with AFL [25] and AFL++ [10] to fuzz Zephyr, specifically the `console/echo` sample. An artificial bug is introduced into the sample code, which will result in a segmentation fault if a certain character is passed to Zephyr. Renode hooks are used to pass data to Zephyr through an emulated UART interface. Custom scripts are required to facilitate the communication between AFL++, Renode, and Zephyr. The setup described in this blogpost describes the alternative way of fuzzing compared to FTZ: emulation. Section 4.1 provides the reasoning behind FTZ's different approach. Compared to FTZ, AFL and AFL++, as used in this project, only support coverage feedback as opposed to FTZ's state feedback. [30]

Finally, Oka and Makila discuss fuzzing Zephyr projects in a continuous integration environment. However, their work is not publicly accessible and the available information about their work does not provide information about technical details of their approach. [31]

3.2 Stateful Protocol Fuzzing

The wider field of protocol fuzzing has received considerable attention. I have selected a set of works published before October of 2024 to represent the potential improvements implemented and evaluated in FTZ. Stateful protocol fuzzing has received considerable attention — refer to the survey papers such as [20], [32], [33] for a more complete overview over the state of the art.

Natella and Pham propose a benchmark for stateful protocol fuzzers iin ProFuzzBench [34], 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. State feedback is provided by HTTP status codes — certain protocols such as FTP already provide these, 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 ($\geq$ 3 participants) protocols cannot currently be fuzzed. Nondeterminism in the programs make 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 [35] is evaluated against this benchmark, while EPF [36] 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 to then be used. Extracting this information from a PUT has been achieved using various techniques across projects:

Manual Annotation Ijon [37] is an extension to AFL and requires analysts to manually annotate the PUT's code. 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 certain variable reaches during execution for the fuzzer to then maximize back 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 successfully trigger the error. There, Ijon showed improved performance compared to unmodified AFL.

Automatic Annotation Manual annotations of state variables is tedious, error-prone, and requires a decent understanding of the PUT internal structure by 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 passing 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. At all locations updating these variables, instrumentation is injected to pass the information back to the fuzzer. [22]

SandPuppy [38] 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 a 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 [23] identifies state-representing memory by looking for variables that are long-lived, 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 smartphone.

Other Greybox Heuristics StateAFL [35] 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 [39] and Ankou [40] take the combination of executed branches as a heuristic for the state of the target. Because this approach leads to a number of states much larger than the states of the abstract state machine powering the server (TCP-Fuzz found 47.9K state transitions in the TCP stack mTCP), Ankou further reduces this complexity for a custom adaptive fitness function.

3.2.2 Use of State Information

5.write something here

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 includes StateFuzz [23], 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 [41] allows users to specify that not only states but also state-transitions should be use to determine the interesting-ness of an input. DDFuzz [42] 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 [22] 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.* [44], who additionally implement FSM minimization. Autofuzz [45] on the other hand employs a bioinformatics algorithm to approximate the FSM.

These structures are then used in several ways: AFLNet [41] uses a custom heuristic based on statistics about each state to schedule the next input to be mutated. SGFuzz [22] 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 further be used to schedule which mutator(s) should be invoked next [44]. ZFuzz [46] 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 [39] has the same target as FTZ: TCP/IP stacks.

Others target BitTorrent [40], DNS [35], FTP [35], [41], [45]–[47], HTTP2 [22], Modbus-TCP [48]–[53], MSNIM [44], RTSP and other media streaming protocols [22], [41], SMTP [35], SSH [35], or SSL/TLS [22], [42], [46] libraries, or SCADA systems [36]. TCP/IP is used by some of these fuzzers to facilitate communication between the fuzzer and PUT, but not evaluated independently or declared as out-of-scope. To compare the effectiveness of improvements introduced by a certain fuzzer, some are further evaluated against mazes [37], [38], levels of the game Super Mario Bros [37], [38], or file formats with complex data-interdependencies [37], [38], [40], [42].

Thus, comparisons are impossible without reimplementing the logic of either related work for FTZ's targets, or the other way around. 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 Modelling and Targeted Specific Mutation

Section 2.2.2 introduced how mutation on inputs that have 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 [36] uses population-based simulated annealing to schedule which packet type to add or mutate next in conjunction with coverage feedback. It requires 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 certain 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 [51], GANFuzz [50] or Autofuzz [45] attempt to automatically learn the structure of the inputs based on generative adversarial networks [50], simplified transformers [51], or bioinformatics techniques [45].

3.4 Machine-in-the-Middle

As opposed to other works, FitM [47] and Autofuzz [45] 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 intelligently mutate messages.

3.4.1 TODO: Read

- *A Survey of Protocol Fuzzing* [32]

4 Implementation

4.1 native_sim

During the compilation of Zephyr, a target board has to be set. One such target provided by the Zephyr project is called `native_sim`, with which the entire operating system and all user code can be compiled into a Linux executable based on a POSIX architecture. 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. [54]. Its integration with Zephyr provides certain functionality to use the host operating system's functionality such as ethernet, UART, or display drivers, among others. [55]

The ability to run Zephyr as a native executable allows the use of native debugging tools such as `gdb` and 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. [56] 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). I worked together with the maintainers of Zephyr to triage and fix this bug. [57]

`native_sim` allows manipulating the clock of Zephyr, to run the operating system faster or slower than realtime, or disabling any speed restrictions. The latter however renders any `sleep`-interrupted 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 an other 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 more details on 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:

Listing 1: SanitizerCoverage Hook Implementations

```c
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;
}
```

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, 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, which is included in the build system. For this, 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 be more error-resistant, the coverage is directly written into shared memory created by the fuzzer. During the initialization, the shared memory is opened based on information passed through environment variables. Additionally, the information needed to map from the SanitizerCoverage-reserved memory section to the shared memory section is stored. On each edge, the appropriate address in the shared memory section is calculated and marked as visited. While there is a slight performance-penalty on each edge due to the address mapping, this approach does not require any post-processing 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 any post-processing function would 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. [55] However, for this project, I opted to build an alternative solution to circumvent the additional limitations of the host kernel interaction. Instead, I built a custom ethernet driver based 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 with 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, the information of which is passed to the PUT using environment variables. There, it is initialized in the driver initialization hook provided by Zephyr. The memory section is split into four parts: one buffer and status field each for each direction. The status field is set to a negative number as long as the buffer is empty, and set to the length of the packet 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 need to 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 Modelling 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 Modelling and Mutation Target

LibAFL provides a composite input type out of the box: `MultipartInput`. It consists of a variable number of entries, and for mutation, a random entry is chosen. This is equivalent to the algorithm for selecting which input message to mutate presented in AFLNet [41] or TCP-Fuzz [39]. 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 similar fashion to FitM [47]. Maier *et al.* essentially build a tree by snapshotting executions and only appending entries. This way, any state can still be reached, but it does not require the fuzzer to needlessly evaluate long traces.

In FTZ, I introduce `ReplayingStatefulInput`. It contains a list of messages, and each entry can be mutated in two ways: either a message can be appended (either generated or extracted from another trace), or the *last* entry is mutated. To achieve this, `ReplayingStatefulInput::map_mutators` automatically maps any mutator targeting the *message* type to one targeting the `trace` type, while also handling empty traces.

4.4.2 Seeding

Since both options of modelling 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 an entire full and legal interaction between a client and Zephyr. The trace is then filtered for TCP messages going from the client to the server. Then, the corpus is seeded with slices of this filtered list of messages, starting with the 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 is able to explore — the empty trace is still part of the corpus. But 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 Modelling 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, (2) 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 modelled does not matter, since the messages themselves are never changes. However, this approach severely limits the kinds of traces a fuzzer is able to generate and 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++ [10]. 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 and thus fuzzers relying on packets generated through `havoc_mutations` are not going to be able to explore past the server's code past the initial parsing logic.

The last option includes modelling 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 number 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 as to not be outright rejected by the parser.

When employing the third option, one has fine control over what mutations are done 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 overriding the mutated values to the correct ones selectively. This approach tests both all parts of the input parser and still is 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, there are libraries available, which significantly reduce the required programming overhead, but one still needs to decide which parts of the message should be randomly mutated and which parts 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 take into consideration 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 `u8` representation of its header flag, other packets are mapped to 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 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, incentives 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 evaluation of a trace. This further improves reliability, with a similar argument as is presented in Section 4.2 .

Section 4.3 discussed 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 both from the client and the server, both directions are taken into account. 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 the state map. State transitions are only measured for each incoming message, because otherwise, when the fuzzer finds a way to send at least two arbitrary packets without a response, the entire state space would be accounted from 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 details of the implementation of the different parts of FTZ. For additional details refer to the architecture diagram of LibAFL in Figure 1 and their paper [26] for the interaction of the different structures.**6.Move figure**

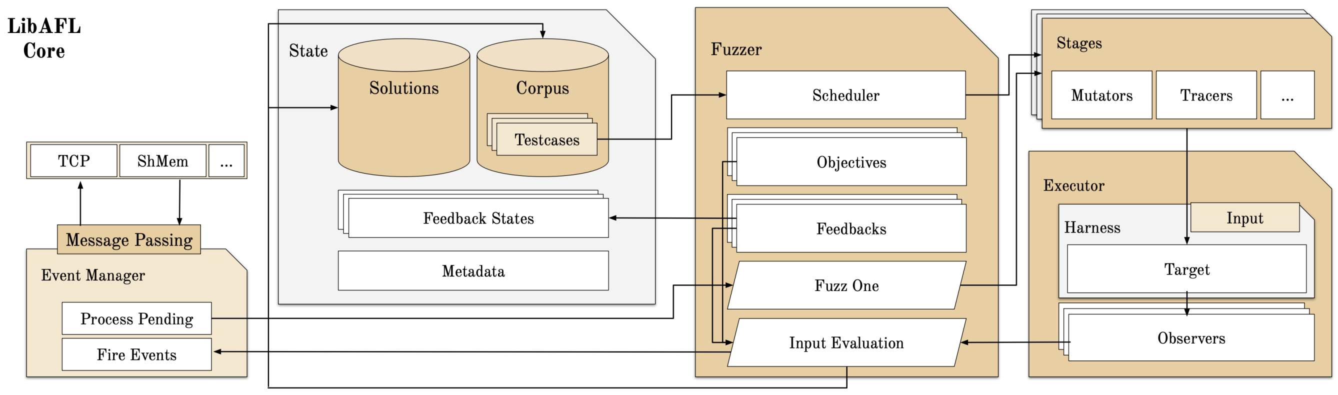

Figure 1: LibAFL's Architecture [26]

4.6.1 Central Infrastructure

The fuzzer relies on `CentralizedLauncher` to spawn a number of clients, each with their own data structures and each spawning their own instance of Zephyr. `CentralizedLauncher` synchronizes inputs that were 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 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 effectively scale across thousands of clients.

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. Additionally, a graphical status-board, and a monitor relaying both a client's logs and aggregated logs to the global logging system are provided. LibAFL further supports more complex logging to different data types or other systems such as a prometheus instance, all of which are easily integrated into FTZ.

4.6.2 Client Setup and Operation

Each client at the start of the fuzzing process initializes its own 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 interesting-ness 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 solutions 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 [24].
- 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++ [10].
- 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 setting up all the required data structures, the corpus is filled based on a recorded trace (refer to Section 4.4.2). Because some of the trace subsets result in the same state coverage feedback, but they are still all useful to the fuzzer as starting points from mutations, they are force-generated and afterwards 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 in, 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 startup, while logging packets emitted by it, and only responding to those that require a manual response (see 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 with 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. These 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 wait states necessary for e.g. synchronization.
- `int_mutators` include the applicable mutators from `havoc_mutations` targeting numeric types.
- `BoolMutator`s flip boolean values.
- `ValueInput`, an improvement to how inputs encapsuling 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 bugfixes related to `MultipartInput`.
- Several general architectural and clean-code improvements.
- `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 `ReplayingStatefulInput`.

[2]Not currently used in FTZ.

5 Evaluation and Results

FTZ was evaluated along multiple axis, starting with experiments on the consistency of the target, 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, Timeouts, and Real Time

To evaluate the consistency of the PUT, I changed FTZ to use `StdScheduler`, which selects inputs from the corpus at random, to generate a wider 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, up to 1500 executions per input.

When FTZ is setup 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 inputs of a certain trace length. The evaluations were run across 64 cores, with 10 clients per core, for 24 hours and approximately 40 million target executions on individual feedbacks, and for 48 hours for the combined feedback.

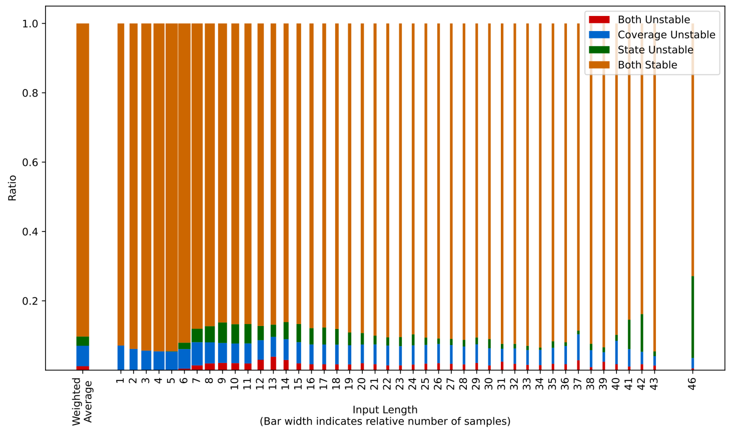

Figure 2: Average stability of both feedbacks across trace lengths

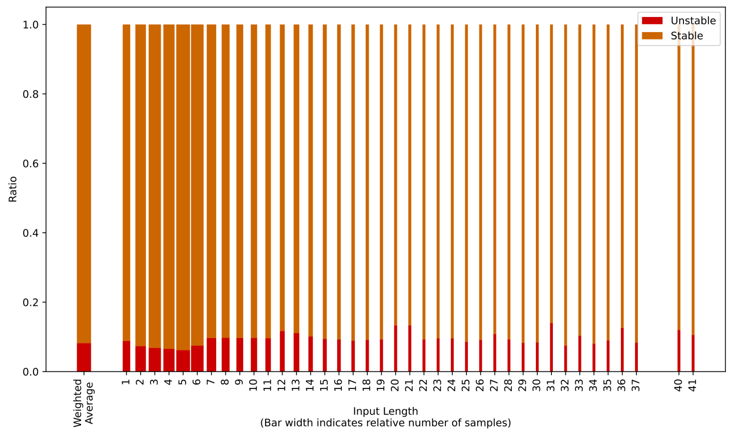

Figure 3: Average stability of coverage feedback across trace lengths

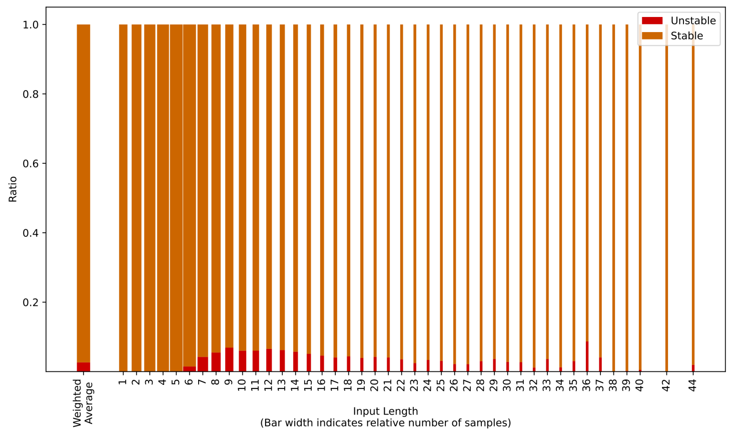

Figure 4: Average stability of state-diff feedback across trace lengths

5.2 Throughput and Overcommit

5.3 Input Modelling and Mutation

With fixed feedback:

- `MultipartInput` and `havoc_mutations()`
- `ReplayingStatefulInput` and `havoc_mutations()`
- `ReplayingStatefulInput` and parsed structure with different mutators

5.4 Feedback

- Coverage (repeatability!)
- State marking
- State diff marking

6 Discussion

6.1 Feedback

- How effective do different improvements seem to be

6.2 Input Modelling and Mutation

- How effective do different improvements seem to be

7 Conclusion

7.1 Future Work

7.move to discussion?

7.1.1 Improved Performance

- This was not prioritized further because memory was the limiting factor
- Semaphores instead of busy waiting

7.1.2 Alternate Targets

- Other OS network stacks
- Userland network stacks

7.1.3 Improved Oracles

- e.g. Differential Fuzzing
- Sanitizers

7.1.4 Improved Scheduling Based on State Feedback

7.1.5 Generilazation of Techniques

7.1.6 Extended Evaluation

- System calls
- IPv6
- Other network protocols
- Comparison to other fuzzers (not done because none were applicable without extensive engineering work to ensure compatibility with target)

7.2 Contributions

- Summary of paper

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.

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] 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`.

[5] 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/`.

[6] 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.

[7] 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).

[8] 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.

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

[10] 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.

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

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

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

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

[15] V. Huber, "Challenges and Mitigation Strategies in Symbolic Execution Based Fuzzing Through the Lens of Survey Papers," Dec. 2023. [Online]. Available: https://github.com/riesentoaster/review-symbolic-execution-in-fuzzing/releases/download/v1.0/Huber-Valentin-Challenges-and-Mitigation-Strategies-in-Symbolic-Execution-Based-Fuzzing-Through-the-Lens-of-Survey-Papers.pdf.

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

[17] 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.

[18] 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.

[19] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence.," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, vol. 19, 2019, pp. 1–15.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] "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).

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

[30] "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).

[31] D. K. Oka and T. Makila, "A practical guide to fuzz testing embedded software in a ci pipeline," *FISITA World Congress 2021 - Technical Programme*, 2021.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] "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).

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

[57] "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).