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

Valentin Huber

at Cyber Defence Campus

and Institute of Computer Science at ZHAW

contact@valentinhuber.me

February 1, 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.

Keywords: Software Testing, Fuzzing, Stateful Fuzzing, Zephyr, LibAFL.

Contents

1 Introduction **2**
- 1.1 Quotes, TODO: Integrate . . . 2
  - 1.1.1 TCP/IP . . . 2
- 1.2 Contributions of this Project . . . 2

2 Background **3**
- 2.1 Fuzzing techniques . . . 3
  - 2.1.1 Generational Fuzzing and Mutation . . . 3
  - 2.1.2 Feedback . . . 3
- 2.2 Fuzzing Network Stacks Is Hard . . . 3
  - 2.2.1 Deep integration with OS . . . 3
  - 2.2.2 Network packets are highly structured . . . 3
  - 2.2.3 TCP Stacks Have an Internal State . . . 3
- 2.3 LibAFL . . . 3

3 Related Works **3**
- 3.1 Protocol Fuzzing . . . 3
  - 3.1.1 Observing State . . . 3
  - 3.1.2 Use of State Information . . . 3
  - 3.1.3 Input Modelling and Mutation . . . 3
- 3.2 Snapshotting . . . 3
- 3.3 Data, TODO: Integrate . . . 3
  - 3.3.1 TODO: Read . . . 5

4 Implementation **5**
- 4.1 native_sim . . . 5
- 4.2 Coverage Information . . . 6
- 4.3 Network Interfaces . . . 6
- 4.4 Input Modelling and Mutation . . . 7
  - 4.4.1 Trace Modelling and Mutation Target . . . 7
  - 4.4.2 Seeding . . . 7
  - 4.4.3 Input Message Modelling and Mutation . . . 7
- 4.5 State Inference Heuristic . . . 8
- 4.6 Implementation Details . . . 8
  - 4.6.1 LibAFL Structure . . . 8
  - 4.6.2 Helper Functionality . . . 8
- 4.7 LibAFL . . . 8
  - 4.7.1 Overcommit . . . 8
  - 4.7.2 Numeric Mutators . . . 8
  - 4.7.3 Mapping Mutators . . . 8
  - 4.7.4 TODO: others . . . 8

5 Results **8**
- 5.1 Consistency, Timeouts, and Real Time . . . 8
- 5.2 Throughput and Overcommit . . . 8
- 5.3 Input Modelling and Mutation . . . 8
- 5.4 Feedback . . . 9

6 Discussion **9**
- 6.1 Feedback . . . 9
- 6.2 Input Modelling and Mutation . . . 9

7 Conclusion **9**

7.1 Future Work . . . 9
7.1.1 Improved Performance . . . 9
7.1.2 Alternate Targets . . . 9
7.1.3 Improved Oracles . . . 9
7.1.4 Improved Scheduling Based on State Feedback . . . 9
7.1.5 Generilazation of Techniques . . . 9
7.1.6 Extended Evaluation . . . 9
7.2 Contributions . . . 9

Bibliography **9**

1 Introduction

Zephyr is an open-source real-time operating system hosted by the Linux Foundation. [1] Real-time operating systems 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, embedded devices such as emergency water detectors. [3]

Zephyr projects typically connect to other devices or the internet using one of the operating system'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.1.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. Ever since its inception in the nominal work by Miller *et al.* [5], 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" [6]) include observing the program execution to guide the fuzzer when selecting the next input to test, or improved oracles such as memory sanitizers. However, fuzzing the network stack of an operating system poses three challenges: (1) they are deeply integrated with the operating system, (2) they require highly structured input data, and (3) they have internal state. These challenges are further elaborated in Section 2.2.

This thesis presents FuZetwork, a proof-of-concept fuzzer using advanced input representation and mutation techniques, stateful feedback, and approaches to effectively test an unreliably PUT targeting the TCP/IP stack of Zephyr. It 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 FuZetwork, along with how they were implemented,
- Section 5 presents the results achieved in this project,
- Section 6 discusses the implication of those results on the effectiveness of FuZetwork, and
- Section 7 summarizes the findings of this thesis.

1.1 Quotes, TODO: Integrate

"The input language of a stateful system consists of two levels: (1) the language of the individual messages, which we will refer to as the message format, and (2) the language of traces, built on top of that. A description or specification of such an input language will usually come in two parts, one for each of the levels: for example, a context-free grammar for the message format and a finite state machine describing sequences of these messages. We will call the latter the state model or, if it is described as a state machine, the protocol state machine." [7]

1.1.1 TCP/IP

Checksums such as in TCP/IP are challenging. [8] Approaches to deal with this include removing the program sections from the target program [9].

1.2 Contributions of this Project

A fuzzer that targets the TCP/IP stack of Zephyr, which

- employs a framework that allows running Zephyr as a native executable, removing the necessity of an emulation or translation layer,
- introduces a custom ethernet driver based on shared memory to make fuzzing parallelizable and performant,
- guides mutation based on coverage information and a heuristic to infer the state of the TCP state machine based on flags in the TCP header of responses from the server

This project further

- evaluates how different ways of using the feedback described above influence fuzzer performance,
- evaluates different ways of modelling the packets passed to the fuzzer,

- evaluates different approaches to mutating those, and
- introduces various performance optimizations to LibAFL

2 Background

2.1 Fuzzing techniques

2.1.1 Generational Fuzzing and Mutation

2.1.2 Feedback

2.2 Fuzzing Network Stacks Is Hard

2.2.1 Deep integration with OS

- Environment/Hardware modelling necessary
- Usually, whole system needs to be run, which is a performance issue
- Instrumentation (Coverage, Sanitizers) is hard

2.2.2 Network packets are highly structured

- Dependencies between data (length fields, checksums, etc.)
- Dependencies between packages (state in TCP)

2.2.3 TCP Stacks Have an Internal State

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" [7]. TCP stacks are such a stateful system. This thesis will further 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." [7]

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. [7] Indeed, stateful targets are listed among the remaining challenges in fuzzing by Boehme *et al.* in their review paper. [10]

This introduces two fundamental challenges on 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, FuZetworkrestarts 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.

2.3 LibAFL

3 Related Works

- Focus on Network Protocol Fuzzing

3.1 Protocol Fuzzing

3.1.1 Observing State

- Manual annotation
- Automatic annotation
- Other greybox approaches
- Heuristics

3.1.2 Use of State Information

- Feedback/Maximization
- Scheduling

3.1.3 Input Modelling and Mutation

3.2 Snapshotting

3.3 Data, TODO: Integrate

- *Stateful Greybox Fuzzing* [11]: "In this paper, we argue that protocols are often explicitly encoded using state variables that are assigned and compared to named constants [...] More specifically, using pattern matching, we identify state variables using enumerated types (enums).

An enumerated type is a group of named constants that specifies all possible values for a variable of that type. Our instrumentation injects a call to our runtime at every program location where a state variable is assigned to a new value. Our runtime efficiently constructs the state transition tree (STT). The STT captures the sequence of values assigned to state variables across all fuzzer-generated input sequences, and as a global data structure, it is shared with the fuzzer." [11] Built on LibFuzzer

- *StateAFL: Greybox fuzzing for stateful network servers* [12]: compile-time probes observing memory allocation and I/O operations; state inference based on fuzzy hashing of long-lived memory areas.

- *Ijon: Exploring Deep State Spaces via Fuzzing* [13]: Manual annotations of code, to manually add entries to an AFL-style map (set/inc at calculated offset), include state information (variable values) in how edge coverage is calculated, and store the max value a certain variable reaches during execution for the fuzzer to then maximize.

- *SandPuppy: Deep-State Fuzzing Guided by Automatic Detection of State-Representative Variables* [14]: Ijon [13], but automatic (initial run capturing variable-value traces, analyze along with source code, add Ijon-style instrumentation, repeat during fuzzing)

- *The Use of Likely Invariants as Feedback for Fuzzers* [15]: run for 24 hours, record variable values and relationships between them, then add a feedback that rewards when the generated assertions are violated

- *Ankou: guiding grey-box fuzzing towards combinatorial difference* [16]: take combination of executed branches into consideration, reduce to manageable adaptive fitness function

- *FuzzFactory: domain-specific fuzzing with waypoints* [17]: framework to add custom feedbacks like number of basic blocks executed, amount of memory allocated, etc.

- *ParmeSan: Sanitizer-guided Greybox Fuzzing* [18]: Use sanitizers checks as fuzzing targets

- *Fuzzing with Data Dependency Information* [19]: Use execution of new data dependencies as feedback

- *StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing* [20]: Find state variables (long-lived, can be updated by users, change control flow or memory access) using static analysis, use that to guide fuzzing (new coverage, new value-range, new extreme value). (Talk: Good Example of why coverage-guided alone is insufficient). Check value ranges instead of all values (static symbex!). 4-digit number of state varia les in linux kernel and Qualcomm MSM kernel (Google Pixel).

- *ProFuzzBench: a benchmark for stateful protocol fuzzing* [21]: Suite of 10 protocols and 11 open-source implementations of those to be tested. TCP is notably absent from this list. Certain protocols (like FTP) already return HTTP status codes, others are patched to do so. Dockerized. The authors note that configuration is not taken into account and multi-party (≥ 3) protocols can not be fuzzed right now. Non-determinism 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. Speed is another issue, where complex setup-processes, costly network operations (resp. synchronization for me), and long multipart-inputs contribute. Finally, state identification is only superficially handled.

- *AFLNET: A Greybox Fuzzer for Network Protocols* [22]: FTP and RTSP as targets, state transition (+coverage) feedback, corpus from traces, mutation on random entry (havoc and insert/delete/duplicate etc.), corpus scheduling based on statistics about each state.

- *TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing* [23]: TCP targets, differential fuzzing, input: syscalls + packets, complex mutators including intelligent dependencies between the current and all previous packets and syscalls in the input, feedback: inter-package diff in coverage.

- *FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols* [24]: Use persistent snapshots of userspace processes (CRIU), fuzzer-in-the-middle, multiple strategies such as input deduplication and resynchronization necessary because of the approach

- *Autofuzz: Automated network protocol fuzzing framework* [25]: Man-in-the-middle, learn protocol by constructing a FSA and packet syntax

using a bioinformatics technique. Fuzz server and client.

- *EPF: An Evolutionary, Protocol-Aware, and Coverage-Guided Network Fuzzing Framework* [26]: Target SCADA, uses population-based simulated annealing to schedule which packet type to add/mutate next in conjunction with coverage feedback. Requires Scapy-compatible implementation of packet types to ensure packet structure.
- *A model-based approach to security flaw detection of network protocol implementations* [27]: Build FSM and minimize it, then use it to schedule basic mutations.
- *GANFuzz: a GAN-based industrial network protocol fuzzing framework* [28]: Learn GAN to generate next inputs. Targets Modbus-TCP.
- *Fuzzers for Stateful Systems: Survey and Research Directions* [7]: Provides taxonomy of components and categorizes stateful fuzzers, compares approaches and lists challenges and future directions.
- *Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach* [29]: Manual FSM from RFC to generate abstract attack models against congestion control implementations, which are then transformed to concrete attacks strategies and tested on actual implementations.
- *A Modbus/TCP Fuzzer for testing internetworked industrial systems* [30] is a Modbus fuzzer, that only seems to use TCP as transport. Generation of packets happens only for Modbus itself. *MTF-Storm: a High Performance Fuzzer for Modbus/TCP* [31] is follow-up work that does the same more systematically.
- *A survey on fuzz testing technologies for industrial control protocols* [32] review industrial control protocol. Notably, Modbus/TCP fuzzing seems common, but TCP is only used as transport layer, not target.
- *MTA Fuzzer: A low-repetition rate Modbus TCP fuzzing method based on Transformer and Mutation Target Adaptation* [33] use machine learning models to generate Modbus/TCP packets, but target Modbus and again only use TCP as a transportation layer.
- *A vulnerability detecting method for Modbus-TCP based on smart fuzzing mechanism* [34] is another Modbus/TCP fuzzer that does not fuzz the TCP stack. See also [35].
- A similar approach using a Scapy-based fuzzer was implemented for the industrial protocol EtherNet/IP, where TCP was declared as out-of-scope [36].
- *Fuzz Testing of Protocols Based on Protocol Process State Machines* [37] calculate a directed graph from the measured state variables, and schedule mutations based on a formula incorporating state depth, coverage, number of transitions, and number of mutations based on this state.

3.3.1 TODO: Read

- *A Survey of Protocol Fuzzing* [38]

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. [39]. Its integration with Zephyr provides certain functionality to use the host operating system's functionality such as ethernet, UART, or display drivers, among others. [40]

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. [41] 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. [42]

`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, FuZetworkmeasures the executed code under each trace (refer to Section 2.1.2 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

```
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 FuZetwork, 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 Network Interfaces

Zephyr's `native_sim` already provides an ethernet driver for a TAP interface over a zeth interface. [40] 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, FuZetworkmanually 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. FuZetworkimplements 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 TCP-Fuzz [23]. 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 [24]. 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 FuZetwork, 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++ [43]. 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.

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.

- Based on responses by server
- Parsing, different non-TCP responses classified
- On TCP packets, the header flags are used

4.6 Implementation Details

4.6.1 LibAFL Structure

- Input Scheduling Algorithm
- Mutation Scheduling Algorithm
- Oracles
- MapObserver/-Feedback
- Executor

4.6.2 Helper Functionality

- Manual connection
- PCAP
- Scripting

4.7 LibAFL

To further simplify this task, I have contributed a range of improvements to LibAFL:

- `MappingMutator`s are essential for this approach — 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]
- `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.

4.7.1 Overcommit

4.7.2 Numeric Mutators

4.7.3 Mapping Mutators

4.7.4 TODO: others

5 Results

5.1 Consistency, Timeouts, and Real Time

5.2 Throughput and Overcommit

5.3 Input Modelling and Mutation

With fixed feedback:

- `MultipartInput` and `havoc_mutations()`

[1] Currently, only non-crossover mutations are supported in FuZetwork, because of a limitation in LibAFL with using nested `MappingMutators`. 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`.

- `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

2.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 of this project is 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] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2829–2846, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi.

[16] 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.

[17] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: Domain-specific fuzzing with waypoints," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360600. [Online]. Available: https://doi.org/10.1145/3360600.

[18] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 2289–2306, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] F. Tacliad, "ENIP Fuzz: A scapy-based ethernet/ip fuzzer for security testing," Ph.D. dissertation, Monterey, California: Naval Postgraduate School, https://hdl.handle.net/10945/56714, Sep. 2016.

[37] 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.

[38] 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.

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

[40] "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).

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

[42] "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).

[43] 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.