Master Thesis

Fallaway:
High Throughput Stateful Fuzzing By Making AFL* State-Aware

Master Computer Science/Cyber Security
Faculty of Eletrical Engineering, Mathematics & Computer Science
University of Twente

Timme Bethe
s2102315

May 29, 2024

Examination Committee:
Continella, Andrea, dr.ir. (University of Twente - SCS)
Huismas, Marieke, prof.dr. (University of Twente - FMT)
Daniele, Cristian, MSc. (Radboud University)

Abstract

Fuzzing is a popular software test technique. Stateful fuzzing refers to testing stateful software, such as the software implementing vital network protocols which keep the internet secure.

This thesis proposes a new approach to code coverage-based stateful fuzzing. We extend a previously proposed fuzzer, AFL*, which is faster than other approaches but unaware of states. Our approach, called Fallaway, is a code coverage-based stateful fuzzer that combines techniques from AFL* and AFLnet. Fallaway is implemented by extending LibAFL, a library to build modular fuzzers written in Rust.

Fallaway outperforms AFLnet in terms of code coverage when fuzzing LightFTP, gaining 16% more code coverage. This improvement largely can largely be attributed to reusing the target process for multiple test cases instead of just one. The state-awareness of the test cases and feedback does not seem to affect the performance in terms of code coverage.

There are limitations to the approach of Fallaway, which if solved might increase the benefit gained from having state-aware test cases and feedback.

Contents

1 Introduction **5**
- 1.1 Fuzzing . . . 5
- 1.2 Stateful fuzzing . . . 6
- 1.3 Our approach: Fallaway . . . 6

2 Background **7**
- 2.1 Stateless Fuzzing . . . 7
- 2.2 Stateful fuzzing . . . 10
- 2.3 AFL*: A different approach to stateful fuzzing . . . 12
- 2.4 LibAFL . . . 14

3 Problem Statement **15**
- 3.1 Turning AFL into a fuzzer for stateful systems . . . 15
- 3.2 Problems of stateful fuzzing . . . 15

4 Fallaway Approach **17**
- 4.1 Clearly separating state scheduling and seed scheduling logic . . . 18
- 4.2 Separating the observations and feedback . . . 18
- 4.3 State scheduling algorithms . . . 18
- 4.4 Increasing test case throughput . . . 19

5 Fallaway Implementation Details **20**
- 5.1 Architecture overview . . . 20
- 5.2 Extending LibAFL for stateful targets . . . 21
- 5.3 Extending LibAFL for socket fuzzing . . . 22
- 5.4 State-aware feedback and test cases . . . 22
- 5.5 Target process reuse . . . 23
- 5.6 Relying on prior knowledge for target state machine information . . . 23

6 Evaluation **23**
- 6.1 Experimental Setup . . . 24
- 6.2 Experiment 1: Comparing Fallaway to the state-of-the-art stateful fuzzers . . . 24
- 6.3 Experiment 2: Varying state-aware feedback and test cases, and target process reuse . . . 25
- 6.4 Experiment 3: Comparing the effect of different state schedulers . . . 26

7 Limitations **27**
- 7.1 Disadvantages of having multiple corpora (Approach) . . . 27
- 7.2 Disadvantages of target state reuse (Approach) . . . 28
- 7.3 Crash triaging (Approach) . . . 28
- 7.4 Difficulties with patching the source code to enable AFL persistent mode fuzzing (Approach) . . . 29
- 7.5 Small sample size (Evaluation) . . . 29
- 7.6 Generality of the approach (Evaluation) . . . 29
- 7.7 Is AFLnet State of the Art? (Evaluation) . . . 30

8 Related work **30**
- 8.1 Code coverage-based stateful fuzzers . . . 30
- 8.2 Man-in-the-Middle Fuzzers . . . 30
- 8.3 State Model Inference and Analysis . . . 31
- 8.4 Machine Learning Fuzzers . . . 31

9 Conclusions 32

10 Future Work 32

1 Introduction

Everyone uses network servers on a day-to-day basis. It is critical to make sure that these network servers are secure. They ensure your data, such as your bank transfer, your medical data or your messages arrive in the correct location, without being changed or read by other parties. Security is critical for network servers because they are connected to the Internet. This connection leaves the servers vulnerable to outside influence. Software vulnerabilities might enable malformed or malicious Internet traffic to cause servers to malfunction or be taken over completely, with loss of data or exposure to other parties as possible results. Thus, it is important to thoroughly test this software for vulnerabilities.

1.1 Fuzzing

One important effort to test software is through a popular technique called fuzz testing or *fuzzing* [18]. Fuzzing is a popular technique to test software where many, often slightly malformed inputs are automatically generated and sent to the software under test, which we call our fuzzing *target*. This popularity can be explained by fuzzing taking relatively little effort to set up yet being effective at finding bugs at the same time. Fuzzing is especially good at finding memory related bugs, which tend to be security relevant, i.e., they are vulnerabilities. The goal of the fuzzer is to find bugs, which it does by finding the format of the input of the target. This enables the fuzzer to create different inputs, which we will call *test cases*, that execute different parts of the target's code. A test case is simply an input created by the fuzzer that is ready to be executed by the target with the goal to execute new *code-coverage*. When we talk about which parts of the target's code has been executed, we often call this the *code-coverage*, i.e., the code-coverage of a test case is the parts of the target's code that were executed when the target processed the test case. This code-coverage is new if the test case executed parts of the code that had not been executed before. The higher the cumulative code-coverage of different test cases – in other words the total code-coverage of the target – the higher the chance a bug is triggered. This is our ultimate goal. To increase the likelihood of finding new code-coverage, and thus finding bugs, modern fuzzers observe the code-coverage of individual test cases. This code-coverage is then compared to the total code-coverage found to see if the test case found new code-coverage. Test cases with new code coverage are stored by the fuzzer to be later mutated into similar but slightly different test cases in the hope to find even more new code-coverage around the same parts of the code that the original test case found new code-coverage. Fuzzers using this approach are often called 'grey-box' fuzzers [20]. While there are other approaches, grey-box fuzzing is the most prevalent [18].

Fuzzing is a mature technology, but we cannot use it out of the box to fuzz network servers because the software used by network servers is often *stateful*. Most fuzzers are meant to fuzz *stateless* software. Software that is stateful reacts differently to the same inputs, depending on its internal state. For example, take a server running an FTP (File Transfer Protocol) server. The server will react differently to a request to transfer a file from the server to the client depending on whether a connecting client has authenticated themselves. Despite sending the same command, an authenticated and unauthenticated client will get different responses because the internal state of the FTP server is different. Another way of defining statefulness is to say that the behavior of the stateful software, here the FTP server, depends not only on the current input, but the current input and all previous inputs of the current session. To take the authenticated and unauthenticated client example again: The unauthenticated client has sent no previous commands before requesting the file transfer, but the authenticated client has sent its username and password, thereby having changed the internal state of the FTP server, before sending that same request. As a result, the different clients get different responses because their preceding messages were different.

1.2 Stateful fuzzing

The reason stateful software is more difficult to fuzz than fuzzing stateless software is that the order of messages needs to be taken into account. For some of the target's code, the target's internal state needs to be just right to get it to execute. For example, to trigger the code in the FTP server that transfers a file from the server to the client, the target state, i.e., the internal state of the FTP server needs to be correct: The client needs to have been authenticated. Another way to explain the same thing, is to say that a fuzzer fuzzing a stateful target needs to not only find the input format, but also the correct order of those inputs. This tremendously increases the complexity and the search space of individual inputs the fuzzer needs to deal with.

To deal with the state in a structured way, most existing techniques for fuzzing stateful targets focus on one target state at a time. To create this focus, they relate a test case to a target state in a *trace*. A trace is simply all inputs send to the target and is built up from a *prefix* and a test case. The prefix consists of all messages except one: The test case. The prefix is meant to change the target's state to the one we want to focus on. The test case is meant find new code-coverage. To illustrate, we take FTP as an example again. If the fuzzer wanted to focus on the authenticated state in FTP, it might execute the following trace with three messages: `user UBUNTU`, `pass PASS1`, *`test_case`*. The first two messages are the prefix and are meant to change the internal state of the FTP server to the state we want to focus on, namely, the authenticated state. The last message is the test case, which could be anything. The fuzzer repeats this process. However, before it can send the next trace, it needs to get the target back to its initial state. Otherwise, the prefix of the trace will affect the target's state in unpredictable ways causing us to lose that focus. To get the target back to the initial state, the software is often simply restarted. The process to get the target back to its initial state, we call *resetting* the target.

Unfortunately, these techniques for fuzzing stateful targets incur huge overhead compared to their stateless counterparts. After every trace that is sent, the target needs to be reset, which is a costly operation. In a performance comparison in the work proposing StateAFL [21], a fuzzer for stateful targets, it is compared to AFLnet [23], another fuzzer for stateful systems. The number of target executions per second tops out around 40, while stateless fuzzers often average at least a thousand executions per second, often many more.

A fuzzer called AFL* [2] takes a fundamentally different approach to stateful fuzzing which does not incur the overhead causing the prefix-based approaches such as AFLnet and StateAFL to be slow.

There is a fuzzer, called AFL* [2], that takes a fundamentally different approach to fuzzing stateful targets and does not incur this overhead compared to the prefix-based approaches above. Every message sent by AFL* is a test case, but the target is only reset after many test cases, think hundreds to thousands, are sent. As a result, AFL* does not send prefixes and needs to reset the target less often.

However, this also means that AFL* has no control over the state that the target is in. Knowing which state the target is in is used by the prefix-based approaches from above to properly divide their attention between the state and to send certain test cases only to certain states. Despite these drawbacks, AFL* reached higher total code-coverage than AFLnet because it can send three orders of magnitude (x1000) more test cases [2].

1.3 Our approach: Fallaway

In this research, we extend AFL* by incorporating techniques from prefix-based approaches, keeping most of their benefits, but not their drawbacks. Our new approach, which we call Fallaway, makes AFL* state-aware while keeping its high test case throughput. As a result, Fallaway can divide its focus between states, knows which test cases are useful in which state

and has much higher test case throughput by sending more test cases before resetting the target.

To combine these approaches, we first broke down the problem of stateful fuzzing into parts: (1) the fuzzer has to divide its focus between states; (2) the fuzzer has to be aware of which test cases should be used in which state, since the effect of the input can greatly differ depending on the target's state; (3) stateful fuzzing is slow. The fuzzer sends many test cases instead of a single one, to decrease how often the target needs to be reset, thereby increasing the fuzzer's speed.

In breaking down stateful fuzzing into parts, we came across a new problem that none of the approaches solve. All approaches only keep track of the total code-coverage to determine if a test case found new code coverage. This makes it difficult to discover code-coverage of a state if another state already explored it.

Therefore, we propose a new fuzzer, Fallaway, that implements these new approaches. Fallaway is implemented in LibAFL [10], a framework to build fuzzers written in Rust. Fallaway manages to reach 16% more code-coverage than AFLnet when fuzzing LightFTP [13]. We will soon open-source our implementation of Fallaway[1].

2 Background

We need a good understanding of the techniques used in fuzzing to properly understand the problems our new approach solves. We also need to understand the current approaches to stateful fuzzing in more detail. As such, we first discuss stateless fuzzing and then stateful fuzzing, i.e., fuzzing targets that hold state. When discussing stateful fuzzing, we first talk about the 'prefix-based' approaches, where a prefix is sent with a single test case after which the target is reset. Then, we discuss how AFL* [2] works in more depth. Lastly, we give an overview of the fuzzer library LibAFL [10]. This overview is needed to understand the details of how Fallaway is implemented.

2.1 Stateless Fuzzing

Fuzzing is a form of automated testing for software, oriented towards security. It functions by feeding the target many automatically generated inputs with the goal of finding vulnerabilities. Different inputs often execute different parts of the target's code, i.e., they have different *code-coverage*. The higher the overall coverage of the target is, the more likely it is to stumble across a bug. When the fuzzer lets the target execute some input with the goal to uncover new code-coverage we call this input a *test case*. In stateless fuzzing, every input executed by the fuzzer will be a test case, but in state*ful* fuzzing, this is not necessarily true, due to the execution in prefix messages. More on that when stateful fuzzing is discussed in detail.

Fuzzers do not generate test cases randomly. The input for most software has a structure. The software first checks if the input is structured correctly, and if this is not the case the input is rejected, or the execution is stopped. For example, an image converter that converts images from `jpeg`-format to `png`-format might stop prematurely if it detects that the input is not a valid `jpeg`-image. This might happen very early on in the execution of the image converter if, e.g., the magic number in the file header is wrong for a `jpeg`, resulting in very low code-coverage. Therefore, to reach deeper parts of the target's code, we want to create test cases that are similar to the structure that is expected. Notice that we said similar, not the same. Having slight deviations from the input format, for example the official specification of the `jpeg`-format is good. These fuzzy parts around where an input is almost correct but not quite are often the inputs that trigger corner cases that the software does not consider.

[1]Implementation will be published here: https://github.com/utwente-scs/fallaway

There are two main approaches to generate test cases that relate to the structure that the target expects as identified by LibAFL [10]. Firstly, the *model-based* approach, which generates test cases according to a model of the input from scratch. This could be based upon human supplied grammar rules, a model inferred from examples of well-formed input or hard-coded into the fuzzer itself. For example, in the case of the image converter as the fuzzing target, we might write code that can generate random `jpeg`-images. Or, we might train a machine learning model on a large set of `jpeg`-images such that it can generate new images that we can use as test cases [8, 16, 28]. Secondly, the *mutational* approach. In the mutational approach, a fuzzer does not generate new inputs from scratch, but mutates existing inputs into new test cases. An input that is mutated to create a new test case is often called a *seed*. For example, we might give a set of `jpeg`-images, i.e., seeds. Ideally, this set of images is diverse, in that their structure covers most of what is allowed in the input format to give the fuzzer a head-start. The fuzzer can then take these images and mutate them by slightly altering, duplicating, removing or adding to parts of the image, to create new test cases.

Another important idea in fuzzing is the use of *feedback*. In essence, the fuzzer is trying the gauge if test cases did well and to use this information to make smart decisions. For example, the fuzzer might choose to mutate a test case that did well, into a new test case in the hope that this new test case also does well. In general, as the LibAFL [10] paper explains, modern fuzzers *observe* information about the execution of some input in the target. Examples of commonly used observations are the time it took to execute the input, or the code-coverage of the input. The fuzzer then employs some kind of logic, called the *feedback* to determine if the test case is *interesting* based off these observations. The word *interesting* is coined by AFL [27], the stateless fuzzer that popularized code-coverage based feedback fuzzing. Interesting can be loosely defined as a test case being worthy of another look. Test cases that are not interesting are immediately discarded. Different approaches might have different definitions of what is considered interesting. To illustrate how this might work, lets examine AFL in more detail and see how it uses observations, feedback, and what it considers interesting.

AFL (American Fuzzy Lop) itself is no longer maintained, but it's successor AFL++ [9], incorporating many improvements and new approaches, is maintained and widely used. One reason for AFL's success is the use of *edge coverage*-based feedback. In other words, AFL used edge coverage as its main observation. These edges are the edges of the control flow graph of the target. A simple example of a control flow graph can be seen in Figure 1. The rectangles are 'basic blocks', a sequence of instructions that are always executed together. Each basic block ends with a conditional jump to another basic block. The arrows are the edges of the graph. Therefore, an observation of the code coverage (edge coverage) of a test case by AFL consists of a list of these edges, namely, the edges that were covered during the execution of the test case.

To enable this observation of the edge-coverage, AFL *instruments* the target code. Instrumenting means that extra instructions are added to the target code to achieve some goal. AFL supplies its own compilers that instrument the code, adding instructions that communicate to the fuzzer which *edges* in the target are taken during execution and roughly how many times. This, along with the time it took to execute the test case are the observations used by AFL.

Now that we know how AFL gets its observations, lets examine how these observations are used by AFL's feedback. To start, AFL keeps track of which edges previous test cases have covered. The feedback logic uses this as follows: If a test case's code-coverage covers edges which have never been covered before, or have never been covered this many times before, a test case is considered interesting. This feedback only considers newness interesting, this approach to feedback is also called *novelty search*. Interesting test cases are added to AFL's *corpus*. The corpus is simply a collection of inputs. AFL uses the mutational approach and uses the inputs in the corpus as seeds to create new test cases. AFL only adds new inputs to its corpus on two occasions: 1) AFL has to be supplied with initial inputs – initial seeds – to start from. 2) Any

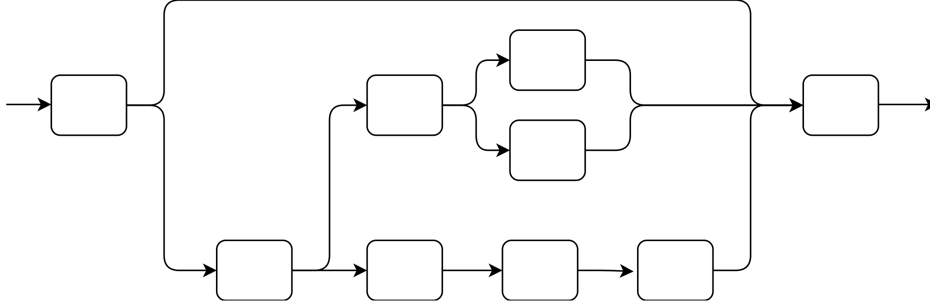

Figure 1: Example of a control flow graph. It shows rectangles, called the 'basic blocks' of a binary program. A basic block is a sequence of instructions that is executed together. Either all instructions are executed, or none are. This is because every block ends with a conditional jump to the start of another basic block. The arrows pointing from one basic block to another are the edges of the graph. These are the edges 'edge coverage' refers too. The edge coverage of an input is the collection of edges that were followed if we were to trace the execution of the program.

interesting test cases are added. This also further explains the use of the word interesting. We do not know if interesting test cases will be useful, but they are worth giving a try.

To recap, AFL uses seeds from its corpus and mutates them to create new test cases. It observes how the test case does and based on this information, feedback logic determines if a test case is interesting. Non-interesting test cases are discarded, but interesting test cases are kept around in the fuzzer's corpus to be used as seeds in the future.

Another important part of the fuzzing puzzle is how a new seed is selected from the corpus. This piece of logic is called the *seed scheduler*. This is an important component because it effectively decides the fuzzer's focus. A fuzzer cannot run forever and for most targets, it cannot try every single input, because there are infinitely many. As a result, a fuzzer has a so-called *time budget*. How this budget is spent influences how well the fuzzer performs and it is decided by the seed scheduler.

AFL's seed scheduler takes the following approach. It computes the minimal set of inputs that together still cover each of the edges in the target, giving priority to small and fast executing test cases. This way, AFL divides its attention between the test cases that are most different – after all, their code-coverage is different. AFL can also be configured to slightly alter the seed scheduler with so-called 'power-schedules', for example 'explore' to find different parts of the code or 'rare' to focus on rarely executed parts of the code. The term power in power-schedule likely refers to the amount of energy and thus time, the fuzzer is scheduling.

Once a seed is chosen by AFL, it goes through several execution 'stages'. This means that in AFL, once a seed is selected, it gets turned into multiple test cases resulting in multiple executions of the target. Examples of stages in AFL are a calibration stage, where the behavior of the seed is examined, a deterministic stage where deterministic mutations are used, a 'havoc' stage where random mutations are stacked upon each other and a 'splicing' stage where different seeds are spliced together to create a new test case.

AFL is used as the base of many other fuzzers and has inspired many other designs. Therefore, understanding how AFL works serves as a good basis to know how most modern fuzzers work. Many other fuzzers use slightly different observations, have different feedback logic, instrument the code in a clever way, have a different seed scheduling algorithm or have different execution stages, but the main components are likely to be the same.

2.2 Stateful fuzzing

By stateful fuzzing, we mean fuzzing targeting stateful targets. Stateful software behaves differently depending on their internal state, which is determined by previously processed input. In other words, the behavior of a stateful system is determined by the current input and all input processed previously.

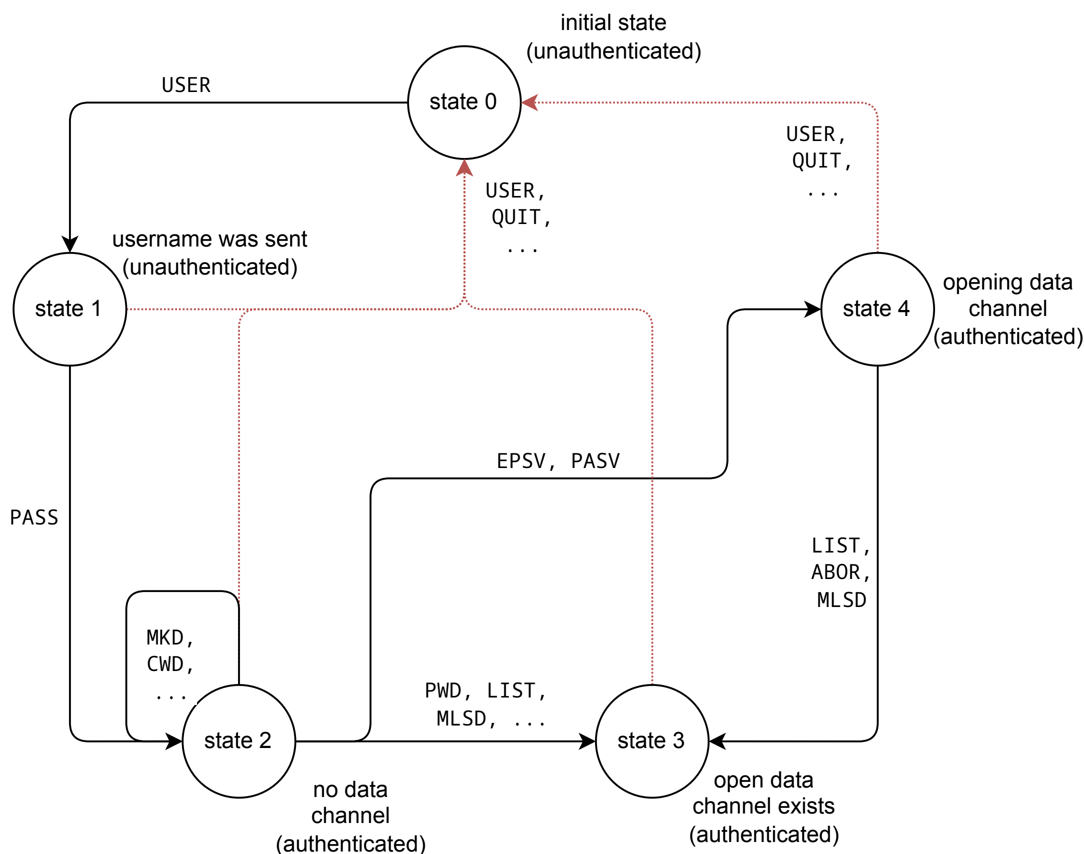

Figure 2: Simplified version of the FTP finite state machine.

Let us repeat the example from the introduction of an FTP (File Transfer Protocol) server, which is a piece of stateful software. In Figure 2 you can see a simplified version of a possible FTP state machine. We interact with the server by sending commands. For example, we could send the `MKD` command to try and create a directory or `CWD` to change the working directory, but also `USER` and `PASS` to specify the username and password to log in with. Certain commands only succeed if the user has logged in, i.e., is authenticated. If we have not previously logged in by sending the correct username and password, sending the `MKD` command not create a directory but will be ignored because unauthenticated users are not allowed to do so. What is important to understand is that the server processes the inputs differently, depending on its internal state.

If we compare how we find code-coverage in stateless vs. stateful targets, it becomes obvious why stateful targets are challenging to fuzz. In *stateless* fuzzing, to find much code-coverage, the fuzzer needs to find the format of the input that the target accepts. In *stateful* fuzzing, to find much code-coverage, the fuzzer needs to still find the format that the target accepts, but it also needs to find the correct order of the inputs. Therefore, we need to explore a combination of inputs and their ordering, massively increasing the total input space compared to stateless fuzzing.

To deal with the ordering of the inputs, i.e., the state, in a structured way, fuzzers need to be

```
1  fuzzer.preprocess(inputs);
2
3  while(running) {
4      trace <- fuzzer.scheduler.next(corpus);
5      fuzzer.mutator.mutate(trace);
6      observations <- target.execute(trace);
7      if (feedback.is_interesting(observations)) {
8          fuzzer.corpus.add(trace)
9      };
10     target.reset();
11 }
```

Figure 3: Pseudo-code of the basic loop of a stateful fuzzer. First the initial inputs are processed, most importantly the initial traces. Then the fuzzing-loop is started: A trace is chosen from the corpus by the scheduler and mutated to create a new trace. This new trace is executed by the target, returning observations. The feedback determines if these observations make the trace interesting, and if so, the trace is added to the corpus.

made state-aware. Most techniques for stateful fuzzing do so by acquiring a list of target states, such that they can focus on one state at a time. Examples of such fuzzers are AFLnet [23], StateAFL [21], SGFuzz [3] and NSFuzz [24]. These fuzzers select one of the states from the list and change the target's internal state to match the selected state, after which a test case is sent. Then this process repeats.

Practically, this happens by *resetting* the target and sending a *trace*. To summarize from the introduction: Resetting a target means to revert it to its initial state – the state it is in when the software starts running. This is often done by simply restarting the software. A trace is simply a *prefix* and a test case. In other words, a trace is a test case related to a specific target state. The prefix messages for a state, or *prefix* for short is a list of messages that when sent to the target, the target ends up in the state that we want. This only works if the target starts out in its initial state, which is why the reset is necessary. Notice how choosing which trace to send, is equivalent to choosing which state to fuzz, because by choosing a trace, you also choose a prefix.

These fuzzers use this prefix-based approach so that they can purposefully divide their focus among the different states. Moreover, having test cases be related to a specific target state is useful too. It means that we can have the same test case for some states, but not for others. In other words, two traces can have the same test case, but a different prefix and a test case can exist for some states while not for others. It makes intuitive sense that we want this. After all, test cases make the target behave differently depending on the target's internal state. As a result, test cases can have different code-coverage depending on the target state. Consequently, test cases might only be interesting in one state, but not another.

Implementation wise, most of these fuzzers are adapted from AFL and thus follow a very similar architecture. The corpus now holds traces instead of individual test cases and the seed scheduler chooses a trace from the corpus – and therefore also which state. Moreover, the mutator knows not to change the prefix of the trace but to only mutate the test case. The feedback mechanism need not be changed. Based on observations, such as code-coverage, the feedback determines if a trace is interesting and interesting traces get added to the corpus.

To illustrate how these these fuzzers function in a different way, a small pseudo-code example of a fuzzing loop is listed in Figure 3. This pseudo-code goes over the same steps as previously explained of resetting the target, choosing a trace, mutating it, executing the trace, and processing the observation as feedback.

One question remains unanswered: How do we get a list of known states in the first place, i.e., what is at least a partial target state machine? This is an important problem in stateful fuzzing but is largely out of scope for this research. Due to its importance, here is a quick overview of how this problem can be solved.

The state machine can be given to the fuzzer beforehand or can be built during the fuzzing process by observing something about the target's state and using this as feedback, i.e., 'state feedback'. Importantly, these methods are not mutually exclusive and can be used to complement each other.

There are multiple approaches to obtain a target state machine beforehand. (1) Active learning is a process where we can choose queries to send to the target and the responses are used to build the state machine [1, 17, 25]. (2) Passive learning is a process where a dataset of request-response pairs is used to build a state machine [4]. (3) The state machine can be human-supplied, for example by deducing it from documentation or other material.

Refining the state model during fuzzing has proven challenging. Different approaches have been proposed, but this is an active point of research [3, 6, 21, 23, 24]. The general idea is to observe behavior related to a specific target state and use this to determine if the target transitioned to a new state after processing an input. Some examples of observations about the state are the responses to the input by the target, as used by AFLnet [23], monitoring long-lived memory regions by StateAFL [21] or monitoring specific variables, such as `enum` types, deemed to be related to the state by SGFuzz [3].

2.3 AFL*: A different approach to stateful fuzzing

AFL* [2] is a fuzzer for stateful targets built on top of AFL++ with a fundamentally different approach to the usual technique with traces explained previously. AFL* can send many more test cases in the same timespan as other stateful fuzzers.

Both approaches are inspired by taking AFL, a fuzzer for stateless targets, and adapting it to be used for stateful targets. However, they do so in a different way.

Let's first get a clear understanding of why AFL cannot be used for stateful targets. Fuzzers for stateless targets only ever send a single input and then wait for the target to terminate after it has processed the input. If this takes too long, the target process is killed; we call this a *timeout*.

This does not work for stateful targets because sending a single input is not enough to be able to cover all code. We illustrate this with another FTP example: If we target an FTP server and we want to execute code in the target that will only run if the target is in a 'client is authenticated'-state, then we first need to send the messages that authenticate the client. In other words, we must send multiple inputs to execute this code.

Moreover, stateful software has the property that it does not terminate after processing the input, instead it waits for further input. In AFL, this will always cause a timeout. As a result, the target process is killed, and the internal state is also reset when a new target process is started. In other words, fuzzers like AFL cannot fuzz stateful targets out of the box, because each test case will time out and they cannot get the target to states other than the initial state.

So, stateful fuzzers need to send multiple inputs. The prefix-based approaches, do this by turning a test case from being a single input, into a sequence of inputs, i.e., a trace. By adding a prefix, we send more messages enabling us to reach different states. After the full trace is sent, the target is reset so the target is ready to receive the next trace.

The AFL* approach does this by sending multiple test cases, and then resetting the target. Let's compare what inputs are sent to the target before resetting its state: The traces look like this, where n is the number of prefix messages, which will never be very large.

$$\text{trace} = \text{prefix msg}_1, \ldots, \text{prefix msg}_n, \text{test case}$$

For AFL*, it looks like this, where m is orders of magnitude larger than n, think m is equal to 1.000 to 10.000:

$$\text{test case}_1, \text{test case}_2, \ldots, \text{test case}_m$$

So, AFL* sends no prefix, but sends many more test cases before resetting the target state. The explains why AFL* is able to send more test cases than AFLnet in the same time span. AFL* resets less often and does not send prefix messages. Resetting the target is a costly operation and time spent resetting is time not spend sending test cases. The same is true for sending prefix messages: Time spent sending prefix messages is time not spent sending test cases.

That sounds great, but AFL* has its has weaknesses. There is a good reason AFLnet sends prefixes, that is because it allows the fuzzer to know which internal state the target resides in. Therein lies the problem with AFL*: Since it does not send a prefix, it has no way to focus its attention between different states. It is still able to transition from state to state because it sends many messages, however, the fuzzer has no control over this. Therefore, AFL* is prone to spend too much time in some states and too little in others.

This weakness comes with another weakness. Just like AFL, AFL* has a corpus with test cases, for which it uses the exact same feedback mechanism as AFL and seed scheduling. It observes the code-coverage and uses novelty search: If a test case discovered new code-coverage, it is added to the corpus. Therefore, AFL* keeps around interesting test cases, but they cannot be related to a target state because AFL* has no concept of state. To repeat, a trace is a test case related to a target state; the prefix defines the state. This means that whichever state the target is in, AFL* will try test cases that are in its corpus because they were interesting when executed in a different state. However, this does not give any guarantee that they are also interesting in the current state.

Let's go over an FTP server example: When the target is in an authenticated state, AFL* sends the `MKD` (make directory) command, which covers new code and is thus added to the corpus. A while later the target is reset, and the target state is the initial – unauthenticated – state. AFL* will still assume `MKD` is an interesting test case, but this is misguided because in this state, the test case is likely to cover the same rejection code as countless other inputs, losing its 'interestingness' in this state.

Another way AFL* is different to prefix-based fuzzers is in its implementation. It introduced the use of *persistent mode* fuzzing for stateful targets. What is persistent mode and how does it work? In persistent mode, a loop is manually patched into the target source code using an annotation, such that it keeps processing new input instead of terminating. This way, a new process need not be forked – which is relatively expensive – for each test case that is executed. This causes fuzzing with persistent mode to much faster than the usual approach of forking a new process for each test case. AFL++'s documentation advises to *only* use persistent mode if the target is state*less*, since otherwise the feedback will become unstable – executing the same input twice might yield different observations – hurting the overall effectivity of the fuzzer[2].

The reason persistent mode can make the feedback unstable is that it can inadvertently turn a stateless target into a stateful one. Imagine some software keeps some temporary state that is normally discarded at the end of processing the input because it would terminate. In other words, the programmer decided to not clean up this state, because they knew the software would terminate, automatically cleaning up the state. Due to our patched in loop, this state is reused when we process the next input. We have inadvertently turned the target into a stateful system.

AFL* is a fuzzer for stateful systems, so its target already process input in a loop, so why do

[2]See here for more info on persistent mode: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md

we need persistent mode? AFL* is based on AFL++, and AFL++ normally expects targets to terminate themselves after processing *one* input else they are killed after a timeout. However, in persistent mode, AFL++ also expects the target to keep processing inputs in a loop. This enables the use of AFL++ for stateful fuzzing when we use it in persistent mode, which is exactly how AFL* is implemented.

An insight by us, which is not mentioned by the AFL* [2] paper itself, is that using persistent mode in this way has another advantage. Most proposed prefix-based fuzzers have the problem that they need to guess how long it takes for the target to process the input. Sending input too fast might cause the input to be dropped due to overflowing buffers, but sending the input too slow means the fuzzer could be more efficient.

This is often solved by setting a generous wait-time, to ensure that the target has enough time. Other fuzzers, such as NSFuzz [24], propose a better approach, namely, to synchronize the target and the fuzzer. Using persistent mode, we this synchronization is built in. To notify that the target has finished processing the input, it suspends its own process. This is an event that the fuzzer can wait for, achieving synchronization.

AFL*'s different approach has merit. It managed to outperform AFLnet [2], a prefix-based fuzzer, in terms of code-coverage. This shows us how sheer execution speed can make up for weaknesses.

2.4 LibAFL

LibAFL [10] is a framework written in Rust to build modular fuzzers. It has components that can be slotted together to create high quality fuzzers that are specialized for their use-case. The components will be familiar because they are similar to those explained in the fuzzing section. Like the name hints at, LibAFL is modelled after AFL.

The motivation for LibAFL is that it greatly decreases the friction in comparing approaches and the friction in code reuse of fuzzers. Currently, many new approaches are implemented with an existing fuzzer (often AFL or some other fuzzer already extended from AFL) as a base. The base is often used unaltered. As a result, any components not relevant to the new approach are left in place causing problems in comparisons whenever they influence the working of the fuzzer.

These fuzzers, like AFL, are often monolithic pieces of software making comparison of approaches and code reuse difficult. There is no modularity. This makes accurately comparing different approaches neigh impossible, because any difference in performance could be caused by any of the differences in the implementations, instead of the approach. Moreover, these extensions are often entangled with the internals of the fuzzer, making reusing the code difficult.

Components of modern fuzzers as identified by LibAFL that are important to understand the implementations details of Fallaway are as follows:

(1) The corpus: Simply a collection of test cases. LibAFL has different implementations such as fully in-memory corpora, or ones that save test cases on disk.

(2) The (fuzzer) state: Every non-volatile piece of data that the fuzzer needs to remember. This includes the corpus, metadata and all state kept by other components such as observers and feedback.

(3) The (seed) scheduler: In LibAFL, this component is simply called a scheduler, but we call this a seed scheduler. This component uses information provided by the feedback to choose which seed from the corpus to use next. For example, LibAFL has a `LenTimeMinimizerScheduler` which prioritizes fast and short test cases and favours a minimal subset of test cases that together cover all code found thus far – just like AFL does.

(4) The executor: The executor provides an interface for interacting with the target. Its main function is to be able to send test cases to the target and return observations about the execution. LibAFL has an executor called the `Forkserver` executor, that makes LibAFL

interoperable with target binaries instrumented by AFL's compilers.

(5) The fuzzer: The fuzzer holds all the logic and glues the components together. It is the bridge between different components and makes sure the right operations happen in the right order. For example, it uses the seed scheduler to select the next seed, invokes the mutator and passes it to the executor for execution. Then it passes on the observations made about the execution to the feedback logic to determine if the test case is interesting and should be saved.

3 Problem Statement

We have discussed that stateful fuzzing is more difficult than stateless fuzzing. While techniques and implementations exist that are tailored to stateful fuzzing, these approaches have weaknesses. On the one hand, we have prefix-based approaches that are aware of the target's state but are slow. On the other hand, we have AFL*, which is fast but lacks focus due to not being state aware.

Ideally, we have a fuzzer that has the strength of both approaches but none of their weaknesses. How can we fuse the two approaches together, to create a better fuzzer? First, we describe exactly what problems arise when turning a fuzzer for stateless systems, such as AFL, into a stateful system. Along the way, we discuss if and how current techniques solve these problems. Then we summarize which properties we want our new fuzzer to have. In other words, which problems should it address?

We create a better fuzzer by identifying specific problems in stateful fuzzing and realizing that no current approach solves all identified problems. This leads us to the following problem statement:

3.1 Turning AFL into a fuzzer for stateful systems

Let us understand how AFL* [2] and AFLnet [23] – a prefix-based fuzzer – turn AFL, a stateless fuzzer, into a stateful fuzzer. Both approaches have the same goal but do so in different ways.

We start by asking: Why can AFL not be used as a stateful fuzzer? The reason is that AFL expects to send a single test case, a single message or input, and wait for the target to terminate. After a certain *timeout*, AFL kills the target to prevent waiting for a target that is stuck in a loop. However, stateful targets expect to process multiple inputs. They wait for input in loop. To give examples, an image converter will terminate itself after converting the image, but the FTP server will not terminate itself after processing a command. Instead, it will wait for a new command to arrive. Thus, if AFL would fuzz a stateful target, all test cases would time out, because the target would wait for new input instead of terminating.

To fuzz stateful software, multiple inputs or messages need to be sent to incite behavior in different states. To incite behavior that can only occur if the FTP server is in an authenticated state, the FTP first needs to be brought to an authenticated state. This authentication, or more generally, the internal state of the target, depends on the messages that were sent previously.

AFLnet, or prefix-based approaches in general, send multiple messages because they send traces, consisting of the prefix and a test case, instead of sending single test case. This enables AFLnet to reach and fuzz different states.

AFL* also sends multiple messages, but instead of a sending a prefix, it sends many test cases. This has the same effect, enabling AFL* to reach different states, gaining speed, but lacking the ability to spread its focus.

3.2 Problems of stateful fuzzing

Now we understand how AFL can be turned into a stateful fuzzer in two ways, each of which tackles specific problems in stateful fuzzing. We identified what problems come up when fuzzing

stateful fuzzing by analyzing the two approaches, leading to a list of four requirements for our new approach. The requirements are listed below. Afterwards, we explain where each of these requirements come from as well if and how they are solved by the two approaches.

I The fuzzer deliberately focusses its energy among target states.

II The fuzzer deliberately focusses its energy within states, not letting test cases from one state negatively impact the focus efforts of another state.

III The fuzzer does not let progress in one state make it more difficult to make progress in another state, due to sharing of feedback information.

IV The fuzzer has higher test case throughput than prefix-based approaches such as AFLnet and StateAFL.

Requirement I: Dividing focus among states Dividing focus is a key problem in fuzzing. AFL divides its time between the seeds in its corpus. The seeds represent the different parts of the code they cover, and AFL wants to spread it's focus between different parts of the code. Being able to focus on different parts of the code is important. Otherwise, the fuzzer would be blind.

This same problem exists in stateful fuzzing at a higher level. In stateful fuzzing, it is important to be able to deliberately divide the fuzzer's attention between target states. Prefix-based approaches can do this, while AFL* cannot. If we know a prefix for a state, we can reset the target and send the prefix to ensure the target is in said state. As a result, whenever a trace is chosen, also a prefix is chosen, allowing prefix-based approaches to deliberately focus on target states. AFL* on the other hand, is blind to which state the target is in and has no way to divide its focus.

Requirement II: State-aware test cases A test case can be interesting in one target state while being simultaneously uninteresting in another state. Therefore, test cases should be related to target states, such that the fuzzer is aware for which states a test case is interesting.

To illustrate this problem, let's go back to the FTP example. If we send a command that requires authentication while the target is in the unauthenticated state, this will trigger code that rejects our command due to the client not being authenticated. The very same code responsible for the rejection, is also triggered by any other command that requires authentication. Thus, only one of these commands is considered interesting in the unauthenticated state. When only concerning the unauthenticated state, the other commands are duplicates: They all cover the same behavior and have the same code-coverage. However, where the authenticated state is concerned, all these commands are interesting as they all have different code-coverage.

If we were to share the test cases between the authenticated and unauthenticated state, this would not be an issue for the authenticated state. After all, all these test cases are considered interesting. The same cannot be said for the unauthenticated state. The fuzzer will spend an excessive amount of time focusing the rejection code triggered by commands that need authentication, when in the unauthenticated state. In conclusion, not relating test cases to a target state can impair the fuzzer's ability to divide its focus *within* a state.

AFL* has a single corpus for all states and thus suffers from this problem. After all, AFL* has no concept of states. However, AFLnet solves this problem by using traces, thereby relating a test case to a state. To reiterate, a trace is a prefix, thus a relation to a state, and a test case.

Requirement III: State-aware feedback Sharing observations and feedback information between states while having state-aware test cases, causes problems with the evolutionary algorithm.

Test cases that are interesting to one state do not have to be interesting for another state, as explained above, but they very well might be. Some code might *only* be covered if the target is in a specific state, but much of the code will be shared between states, and thus test cases covering that code will be interesting for multiple states.

We repeat from the previous identified problem, that having state-aware test cases essentially means that each target state has its own collection of interesting test cases. Consequently, we need to determine if a test case is interesting for each target state separately. This is not possible if we share the observations and feedback information.

We repeat from subsection 2.1, the background on fuzzing, that in novelty search the fuzzer keeps track of a list of edges that are covered previously. If a test case has has code coverage with edges not in this list, it is considered interesting.

Let's assume we have a test case that is interesting for state A and state B. If we execute the test case in state A, the shared list of edges that have been covered gets updated to include the new edges discovered by the test case. If we execute the test case in state B and it discovers the same edges due to shared behavior, the test case will not be considered interesting.

To conclude, if we use state-aware test cases, but do not use state-aware feedback, the fuzzer will have difficulties discovering shared behavior for any state but the first state that discovered it.

AFL* does not have state-aware test cases and thus does not suffer from this problem. Obviously, not having state-aware test cases is a problem of its own. AFLnet does have state-aware test cases due to the use of traces but shares the feedback information between all states. In particular, it shares the list of discovered edges between all states. Therefore, neither approach takes this into account.

Requirement IV: Test case throughput Fuzzers have a time budget. In that time, the fuzzer wants to find as many bugs as possible. To achieve this, the fuzzer needs to try as many test cases as possible. An increase in test case throughput that does not affect other parts of the fuzzer is, therefore, an unequivocal improvement. In some cases, such as AFL*, significant increases in test case throughput can cause a fuzzer to outperform fuzzers that employ smarter techniques, such as AFLnet. In other words, test case throughput is important and faster techniques can perform better than 'smarter' techniques.

Unfortunately, in prefix-based approaches this is throughput low. As mentioned in the introduction, AFLnet and StateAFL have very low test case throughput, 10x-100x slower that AFL fuzzing stateless targets.

While it can be argued that stateful software tends to be more complex, this does not explain the extreme decrease in test case throughput. Moreover, AFL* proves that it is possible to get much higher test case throughput. Thus, this is not an inherent problem to fuzzing stateful software, but a problem that can be improved upon.

4 Fallaway Approach

We have set the goal of the research, namely to create a fuzzer that solve the four mentioned problems. A high level overview of our approach, which we call *Fallaway*, can be viewed in Figure 4. We begin by processing the initial inputs for the fuzzer, for example, a list of prefixes and initial seeds. Then we have two loops. An outer target state level loop and an inner test case loop. The outer loop selects the next state to focus on and ensures the target is in this state. Then, in the inner loop, n number of test cases are sent to the target. We take inspiration from the existing approaches where possible, namely AFL* and AFLnet, which already solve some of these problems.

For each state, we need a prefix to get our target to this state, but we completely decouple prefixes from test cases. Instead, for each target state we have a single prefix and an accom-

```
1  s.preprocess(inputs);
2  while(running) {
3      ts <- state_scheduler.next();               # state loop
4      target.reset_state();                       #
5      target.send_prefix(ts);                     #
6
7      for _ in (0 .. n) {
8                                                  # test case loop
9          seed <- s.seed_scheduler.next(ts);      #
10         t <- mutate(seed);                      #
11         ob <- target.execute(t);                #
12                                                 #
13         s.process_feedback(ob, ts);             #
14     }
15 }
```

Figure 4: Overview Fallaway fuzzing algorithm. `s` is the fuzzer's state. `ts` is an id identifying a target state. Attached this id is a prefix, a corpus and any data stored by the feedback process. `t` is a test case and `ob` are observations, in Fallaway's case most importantly edge coverage.

panying corpus with all test cases for that state. To make our feedback state-aware, we also split up observations and feedback, keeping track of them per state. Lastly, to increase the test case throughput, we take AFL*'s approach: After sending the prefix, instead of sending a single test case, we send many. Next, we will talk about these design decisions in detail, as well as which consequences they have.

4.1 Clearly separating state scheduling and seed scheduling logic

We have seen that using traces enables fuzzer to solve both problem I and II. Namely, it enables the fuzzer to focus on states and to relate a state to the test case. How this attention is divided is discussed in section 4.3.

Our approach clearly separates the solutions to these problems. Our fuzzer has a list of states, each with a corresponding prefix. Each of these states has its own corpus, i.e., its own collection of test cases.

To focus on a particular state, the fuzzer selects a state from the list, this proces we call *state scheduling*. Then, the fuzzer selects a test case from the corpus of the selected state, this process we call *seed scheduling*. Clearly, we are able to divide our attention between states and the interesting test cases from other states do not interfere with the seed scheduling of the selected state.

4.2 Separating the observations and feedback

Although we cannot take inspiration from other approaches solving this problem, the solution is intuitive. Instead of storing the information about observations and feedback, such as the list of covered edges, once, effectively sharing it between states, they are stored together with the appropriate state. When processing observations and feedback when focusing on state A, we store that information with state A. This solves problem III.

4.3 State scheduling algorithms

While our approach is now able to divide its attention between target states, we have not talked about how to divide this attention, which is also part of problem I. The first approach is to

divide this attention evenly among target states. However, there is no reason to think that each state gives us access to the same amount of code coverage. As an example, let's take FTP as our target. The unauthenticated state might simply reject all commands except for those related to logging in. This is likely to result in relatively little code, while an authenticated state has to encode the logic to process many more commands, resulting in more code. Thus, we expect that evenly dividing our focus is not the best option. Instead, we expect that spending more time in the authenticated state compared to the unautenticated state will yield better results.

Therefore, we try two heuristics and their combination to guide our state scheduler. Firstly, we try 'state level novelty search'. We record how many new edges have been covered the last time a state was selected, which is only possible because we have state-aware feedback. The goal is to choose the state that provide more progress, to be chosen more often. Our metric for progress will be code-coverage. To achieve this, we choose the next state based on how many edges were found the previous time that state was selected. For example, we have states A, B and C. The last time A, B, C were chosen, they found 2, 4 and 1 edges respectively. We want to ensure that a state can still be chosen even if it found no new edges the last time, therefore the number of found edges is incremented before being used as weights. As such, the probabilities are 30%, 50% and 20%, respectively. Let's summarize this as (state, weight, probability), for example (A, 2, 30%). Suppose state A is selected. After state A is fuzzed in the inner loop, it happened to find 9 new edges, then the new situation is as follows: (A, 10, 67%), (B, 4, 26%), (C, 1, 7%).

Secondly, we try to guess the complexity of the state according to the target state machine. The assumption is if the current state has many outgoing edges, i.e., there are many ways to transition to other states, that the current state is more complex. We assume that complexity requires more code and complex code is more likely to have bugs. Both are reasons to spend more time fuzzing such states. As a result, we use the same weighted choice as in the novelty search approach above, but we use static weights, namely, the number of outgoing edges of each state.

Lastly, we combine the two heuristics. Finding no new edges for each state in the novelty search heuristic results in an even distribution, but we hypothesize that our second heuristic might be better than the even distribution. Therefore, we have a fourth approach that uses the outgoing edges distribution as a fallback in case this happens. In other words, if the fuzzer found no new edges in all states found any edges the last time they were fuzzed, i.e., (A, 1, 33%), (B, 1, 33%), (C, 1, 33%), we use the outgoing edges as weights.

This results in 4 possible state schedulers, Cycling (CY), Outgoing Edges (OE), Novelty Search (NS) and Novelty Search with a fallback on Outgoing Edges (NO):

- (CY) Target states are selected in a cycle, i.e., in a round-robin type manner. The attention is divided evenly.
- (OE) Outgoing Edges. The distribution of selected states is solely based on the outgoing edges of the target state machine.
- (NS) Pure novelty search. If none of the states made any progress, i.e., found new edges, the distribution is even.
- (NO) Novelty Search with fallback on Outgoing Edges if no state made any progress.

4.4 Increasing test case throughput

To increase the test case throughput and thereby solving problem IV, we reuse the target process. Remember that the only chance to find new code coverage is when the fuzzer is executing new test cases, in other words when the fuzzing is sending new inputs. Although prefix messages are also inputs that sent to the target, they are not new and thus we already know their code coverage.

If we cut down on the time spend doing anything but sending test cases, we will increase our

test case throughput.

In the case of prefix-based approaches, for each test case that is sent, on average multiple other prefix messages also need to be sent as well as resetting the target. For example, if we want to send a test case when fuzzing an FTP server to an authenticated state, we must send at least 3 inputs. The first two inputs are the prefix consisting of the first input indicating the username and the second indicating the password. The third input is the test case. Before the next prefix can be sent, the target needs to be reset. AFL* [2] identified resetting the target as being an expensive operation.

Therefore, prefix-based approaches spend the majority their time resetting and sending prefix messages instead of sending test cases.

The reset and sending of the prefix correspond to line 4 and 5 in Fallaway's general fuzzing algorithm, see Figure 4. The solution Fallaway employs is to send n test cases to the target, before resetting the target and choosing a new state, with n being relatively large, say 10—1000. As a result, line 4 and 5 are executed less often, resulting in more test cases being sent in the time frame. AFLnet discards the target process after a single test case, but we effectively reuse it for many test cases. Clearly, the higher n is, the fewer times we must reset, the higher our throughput will be.

This increase in throughput comes at a cost. AFLnet discards the target process for a good reason: feedback accuracy. Sending any input to the target might change its internal state. In other words, the cost we are paying for higher throughput is losing the certainty of knowing what state the target is in.

Concretely, we make the following trade-off. When we are reusing the target process, there are two options: (1) The test case did not cause a state transition. This is purely a speed increase and there is no cost. (2) The test case caused a state transition. If we are unaware of it, which we will be in our implementation, see subsection 5.6, our feedback will affect the corpus related to the *selected* target state, which does not relate to the actual internal state of the target due to the state transition. In other words, the feedback and observations will be attributed to the wrong state. Moreover, if the next test case is found interesting, it is also added to the *selected* target state instead of the actual target state.

We hypothesize that this trade-off is worth it. We think that the increase in throughput outweighs the disadvantage of sometimes attributing test cases, observations, and feedback to the wrong state. Moreover, there are solutions to this problem, which we discuss in the limitations.

5 Fallaway Implementation Details

Fallaway is implemented in Rust, built on top of LibAFL [10]. This section talks about multiple types of state. To avoid confusion, we will use the following convention in this section: There is the target state: The internal state that our target is in; and fuzzer state: The state `struct` that stores all non-volatile information of the fuzzer. When state is mentioned, the fuzzer's state is meant. When we want to refer to the target state in this section, it will explicitly be called the target state.

5.1 Architecture overview

In Figure 5 an overview of the architecture is shown. It is divided into three main parts: On the left is the state. All non-volatile data used in the fuzzing process is stored here. It stores metadata relevant to all target states, the solutions – test cases that fulfill our objective, namely inputs causing a crash – and it stores a list of 'InnerStates', one for each target state, that store state that is specific to a target state. Specifically, for each target state we store the prefix messages, the test case corpus and metadata stored by components. LibAFL stores any

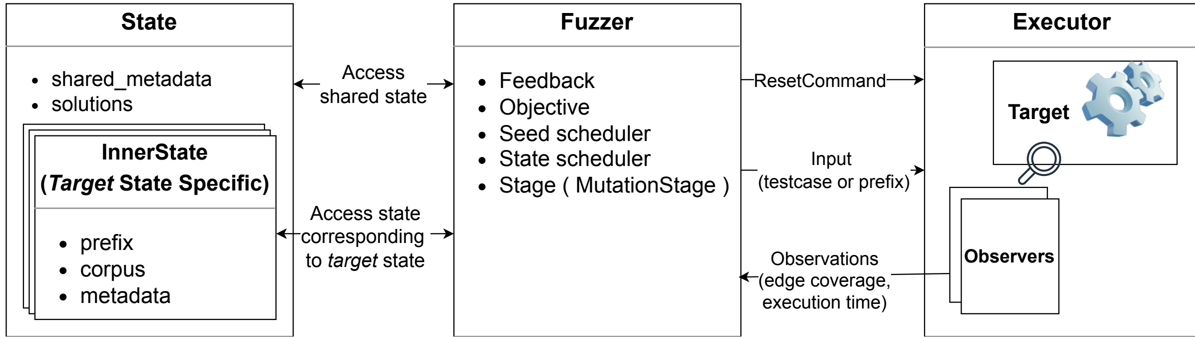

Figure 5: Fallaway architecture overview

non-volatile data, i.e., state, from logic components such as the feedback and the observers as 'metadata'.

On the right is the executor, this handles everything regarding the execution of our target. It also holds the observers, which trace and pass along information about the execution of test cases on the target. In our case, this is this the edge coverage and the execution time.

Lastly, the fuzzer in the middle. It performs all fuzzer logic, where some logic is deferred to components it holds, such as the feedback and both schedulers. The feedback component determines how information from observers is interpreted to decide if a test case is interesting. The objective component is similar but decides if a test case reached our goal. In our case, our objective is crashing the target. The seed scheduler is responsible for selecting the next seed out of a corpus, whereas the state scheduler selects which state is fuzzed next. Lastly, the stages determine how a seed is used, once selected. Our implementation only has a mutational stage, where the seed is mutated into a single new test case and executed.

5.2 Extending LibAFL for stateful targets

We use LibAFL as a base for Fallaway. To repeat, LibAFL is a library to build modular fuzzers. However, LibAFL is built with stateless fuzzers in mind and lacks functionality to deal with stateful targets.

LibAFL supports many targets through various executors to support fuzzing different kinds of targets. Our implementation specifically extends the LibAFL `ForkserverExecutor`. This is an executor that supports targets created by the AFL compiler. Targets that work for AFL and its derivatives such as AFL++ and AFL* can be fuzzed using this executor. This means it also supports AFL's persistent mode, which is why we chose to extend this executor.

To support stateful targets in LibAFL we have to make two major changes. In subsection 3.1, we explained that AFL cannot be used for stateful fuzzing because it expects the target to terminate itself. LibAFL has the same problem. We utilize AFL's persistent mode to change this expectation, as is done in AFL* [2]. In persistent mode, the target is patched such that it processes test cases in a loop without resetting its state. In case of network servers, this often requires little changes because they already accept input in a loop through their send-receive loop.

Secondly, the fuzzer needs to understand the concept of states. We achieve this by introducing an outer fuzzing loop, as in Figure 4. The current implementation of the fuzzer logic in LibAFL only has a test case loop, this is the inner loop. The outer loop selects a state using the state scheduler, resets the target state using the executor and ensures the prefix is sent such that the chosen state corresponds to the state of the target. Then the LibAFL fuzzer component is invoked to perform the inner loop. When the inner loop finishes, we loop back to the start of the outer loop and select the next state.

5.3 Extending LibAFL for socket fuzzing

Most stateful targets are network servers. They communicate over network sockets, i.e., they expect to receive their input and transmit their output over network sockets. This means that test cases also have to be communicated over a network socket. This is not something LibAFL currently supports.

Our implementation can work in both server and client mode. If the target is a network client, the fuzzer starts listening on a socket before starting the target, allowing it to connect to our fuzzer. If the target is a network server, we start the target and connect it on the port it started listening on. Whenever the target is killed, i.e., when a test case takes too long to execute; we reset the target state; or some other connection error occurs, we need to reestablish this connection.

Previous stateful fuzzers such as AFLnet struggled with synchronising the fuzzer and the target. Going too fast risks overloading the target with input which might lead to dropped packets or similar issues, but going too slow means time is spent idling.

Our solution uses the persistent mode as a synchronisation mechanism. When the target is done processing a test case, it stops its own execution, sending a Linux signal to which the fuzzer can wait for. This notifies the fuzzer that it can send the next test case, without wasting time nor having to worry about overloading the target.

There are other options to communicate test cases to the target that do not use network sockets. For example, other options would be to alter the target's source code to make it accept input from a file, standard input or a shared memory buffer. These options can be significantly faster, because the test cases do not have to go through the Linux network stack. In case of a shared memory buffer, the test case communication can be done solely in user space, resulting in the best performance. However, these options require invasive and careful changes to the source code. We stick to using sockets to limit changes to the target source code.

5.4 State-aware feedback and test cases

While the approach to create state-aware feedback and test cases sounds intuitive, implementing it in LibAFL is not.

What needs to happen is that the seed scheduler must use the corpus related to the currently selected state and the feedback component should not use state related to a different *target* state to determine if a test case is interesting.

However, we want to reuse the high-quality components LibAFL supplies. For example, LibAFL supplies implementation of feedback, observers, corpora, and seed schedulers – simply called schedulers in LibAFL. If we want to reuse these components, we cannot change them in any way. Consequently, we cannot explicitly change the way they interact with the fuzzer state.

These components are passed a mutable reference to a struct, the fuzzer's state and interact with this calling certain functions such as `get_corpus()` or `get_metadata()`. We cannot change the component to interact with the state differently depending on the selected *target* state.

Instead, we reimplement LibAFL's state component such that we can switch transparently between *target* states. Any component accessing its state will access its state related to the target state, but from the perspective of the component, nothing changed. In other words, the state component knows which state is selected and can dynamically return different references whenver a component asks for access to its state. To refer to the architecture overview, whenever a component accesses its state, it accesses the state in the InnerState corresponding to the current target state.

Important to note is that these LibAFl components were not designed to be used in this way. They were designed with the assumption to always receive the same reference when accessing their state. This approach only works, if these components do not hold state of their own,

which for the most part they do not. Most components store exclusively store their state in the ‘metadata’ component of the fuzzer state.

As a result, we only have to reimplement LibAFL’s state component, to enable the reuse of most other components.

5.5 Target process reuse

To implement the approach to speed up the fuzzing process, we reuse the target process by sending multiple test cases after the prefix, instead of a single one, this is done using the inner loop.

Implementation wise, this is trivial. We already modified the target to not be reset after reach test case, by using AFL’s persistent mode. In the fuzzer logic, instead of sending a single test case, we simply send n using the LibAFL fuzzer component, as was also explained in subsection 5.2.

5.6 Relying on prior knowledge for target state machine information

There is one question remaining that needs answering. How does Fallaway get access to prefixes in the first place? Prefixes must be supplied to Fallaway as input.

This decision was made to (1) simplify the implementation, experiments, and validations and (2) to narrow the scope of the research.

While using state feedback to build and refine the target state machine is an important problem in stateful fuzzing, LibAFL has no implementation of this either. Implementing this ourselves was not feasible within the time span for this research.

Instead, we rely on prior knowledge about the target’s state machine to be supplied to the fuzzer and do not refine this information during fuzzing.

Again, we stress that this change was made from a practical point of view. Adding a state feedback mechanism to Fallaway increases the ease of set-up, since the fuzzer will figure out the state machine itself. Also, inaccuracies between the implemented state machine and the supplied state machine can be identified and fixed, which could improve the fuzzer’s performance.

Moreover, having accurate state feedback will likely decrease the disadvantages brought forth by our approach to increase the test case throughput. More information about this can be read in the limitations; section 7.

6 Evaluation

To evaluate our approach, the main question to answer is whether Fallaway performs better or worse compared to state-of-the-art stateful fuzzers. Apart from this question, we investigate to what extent the different parts of the approach affect how Fallaway performs. We use the total code coverage (edge coverage) as our performance metric, since finding more code coverage is considered better. To repeat, the chance of discovering a bug becomes larger with higher code-coverage.

To perform this evaluation, we first ask the following main question:

(A) How does Fallaway’s performance compare to the state-of-the-art of stateful fuzzers?

Afterwards, we ask the following questions to investigate to what extent the different parts affect the performs of Fallaway:

(B) To what extent does having state-aware test cases and feedback influence the overall performance of the fuzzer?

(C) To what extent does increasing the test case throughput in stateful fuzzers through process reuse influence the overall performance of the fuzzer?
(D) To what extent do new state scheduler algorithms influence the fuzzer's overall performance?

6.1 Experimental Setup

The experiments are run on a virtual machine with 8 vCPU cores (Intel Xeon Silver 4110 @ 2.10GHz) and 32 Gb of RAM. Each experiment is bound to a single core and the fuzzer is run for 24 hours.

To instrument targets, we use the latest version of AFL++ [9] (commit `775861e`) at the time of writing, with LLVM 14. We use the `afl-clang-lto` compiler from AFL++.

We use LightFTP [13] as our target. This is an FTP implementation that is also used in Profuzzbench [22]. Profuzzbench is a protocol fuzzer benchmark. As a result, we can easily run AFLnet on LightFTP. To enable comparisons with AFLnet, we use the same version as Profuzzbench uses (commit `5980ea1`).

6.2 Experiment 1: Comparing Fallaway to the state-of-the-art stateful fuzzers

To answer question (A), the comparison with the state-of-the-art, we use Profuzzbench [22] to run AFLnet on our target for 24 hours and compare the results. We repeated this four times, the results of which can be found in 6 This is our reference for the state-of-the-art.

Although both Fallaway and Profuzzbench report a code coverage percentage, they are not directly comparable. The total number of edges that can be found differ between Fallaway and Profuzzbench. Fallaway uses the total number of edges instrumented by the AFL compiler, whereas Profuzzbench uses `gcovr` [12], a tool to generate code-coverage reports, resulting in a different number of edges. For this reason, we run the queue from AFLnet, consisting of all traces that found new code coverage, through Fallaway. As a result, they are processed in exactly the same way, and we have a comparable coverage percentage between Fallaway and AFLnet.

Cov(%)	traces in 24 hrs.
34.5	525.000
33.5	476.000
35.5	484.000
33.3	506.000

Figure 6: Results from running AFLnet on LightFTP. Coverage percentage converted to Fallaway. Traces are the number of traces AFLnet sent in 24 hours.

The runs of AFLnet result in an average code coverage for LightFTP of 34.2%. Fallaway was run in different configurations, see Figure 7. Exact information about these configurations is supplied in the next experiment.

Apart from the first 4 configurations, Fallaway seems to outperform AFLnet in terms of code coverage, peaking at 39.7% code coverage, a 16% increase over AFLnet.

In conclusion: Certain configurations of Fallaway outperform AFLnet when fuzzing LightFTP when it comes to code-coverage.

Config	Cov(%)	exec. in 24 hrs. (× mln)
mcmm-cy-1	32.2	9
mcmm-no-1	32.3	10
mcsm-no-1	32.3	9
sc-no-1	32.2	10
mcmm-cy-10	38.2	32
mcmm-no-10	38.0	29
mcsm-no-10	36.3	29
sc-no-10	38.0	29
mcmm-cy-100	39.7	80
mcmm-no-100	35.3	55
mcsm-no-100	37.1	71
sc-no-100	39.6	102

Figure 7: Results from running different configurations of Fallaway on LightFTP. Coverage is the percentage of edges found from the total number of instrumented edges. Last column is the number of executions in 24 hours

6.3 Experiment 2: Varying state-aware feedback and test cases, and target process reuse

To answer question (B) and (C) we run Fallaway in different configurations, varying two variables. The first variable is how state-aware the fuzzer is. Our test cases can be state-aware, in which case we have multiple corpora; one for each target state instead of a single corpus. Also, our feedback can be state-aware, in which case we have multiple metadata maps in our implementations (this is where the feedback information is stored); one for each target state. This leaves us with three versions of state awareness.

mcmm Multiple corpora and multiple metadata maps, i.e., both state-aware test cases and state-aware feedback.

mcsm Multiple corpora but a single metadata map, i.e., each target state has its own corpus, but they share the feedback information. This is also what AFLnet does.

sc Single corpus (and an implied single metadata map). It does not make sense to have multiple metadata maps, because the test case will always get added to the same corpus.

The second variable is the extent that we reuse the target process. This is dictated by n, the number of test cases we send each before resetting the target and selecting the next target state. In our implementation n is called the number of *loops*. We vary the number of loops between $1, 10, 100$. More loops means the target process is reused more. Also note that a loop size of 1 is the case where there is no reuse of the target process at all.

The last component of the configuration name is the middle part, which is the state scheduler, as discussed in subsection 4.3. Putting it all together, the configuration name consists of first the state-awareness, then the state scheduler, followed by the number of loops.

In Figure 7 the results from the experiment are shown. It shows the coverage percentage, i.e., the percentage of edges in the program that were found, and how many executions were performed in the 24 hours; the throughput.

The most striking difference is in the number of loops. Those configurations that have a more than a single loop, namely 10 and 100 loops have significantly higher test case throughput and higher code coverage. However, the difference between 10 loops and 100 loops seems either insignificant or small. Some configurations with 10 loops outperformed those with 100 loops and vice versa.

However, the state awareness of the configuration does not seem to play a role in the code coverage Fallaway is able to find. The state-awareness is expected to have the largest effect when there is only a single loop, since the target process reuse does not cause any inconsistencies in the feedback due to state transitions caused by test cases. However, the code coverage results are the same. This is surprising. We expected *mcsm* to be the worst, since it should be difficult for this configuration to discover shared behavior for all target states, yet there seems to be no difference.

In conclusion: To answer question (B), the state-awareness of the test cases and feedback do not seem to contribute to its ability to discover code coverage. To answer question (C), reusing the target process to send more test cases seems to significantly improve Fallaway's ability to find code-coverage. However, it is unclear what an optimal number of loops is, since the difference between 10 loops and 100 loops is unclear.

6.4 Experiment 3: Comparing the effect of different state schedulers

To answer question (D), we investigate how well our novelty search state scheduler algorithm works. We compare the four different approaches. A quick reminder of the four different algorithms:

- (CY) Target states are selected in a cycle. In other words, in a round-robin type manner.
- (OE) Outgoing Edges. Based solely on the outgoing edges of the target state machine.
- (NS) Pure novelty search. If none of the target states made any progress, a random state is chosen.
- (NO) Novelty Search with fallback on Outgoing Edges.

We use the multiple corpus version of Fallaway with 1000 loops for this experiment.

The results of the experiment can be found in Figure 8. Here, we see the percentage of time spend in different states and the familiar coverage percentage and execution speed. For LightFTP, Fallaway was supplied with the knowledge of all 5 states in the state machine. These states correspond to the state machine in Figure 2, but the state transitions are simplified in the figure.

Surprisingly, NO and OE have the exact same distribution, as do NS and CY. In hindsight, this makes sense. Most of the time, no new coverage is found, so NO falls back on OE. Most of the time is an understatement. Coverage is found so sparsely, that NO practically is OE.

The same is true for NS and CY. With NS, the fallback is effectively CY. CY cycles and the fallback for NS is an equal probability for each state, which result in the same distributions. Therefore, we continue the discussion with only OE and CY, since the novelty search has no effect.

CY seems to perhaps outperform OE, but from this data no hard conclusions can be made. This could be explained by either outgoing edges not having a correlation with the complexity and the number of edges of a target state, or it being correct but the coverage to be found in certain states is exhausted due to LightFTP being relatively simple, after which it would be better to focus on other target states.

In Figure 9 we see the coverage per target state over time for mcmm-ns-1000. The outgoing edges of the states, 0, 1, 2, 3 and 4 are 2, 4, 3, 5 and 6 respectively. If we take Figure 9 to be a heuristic of the total coverage percentage to be found in each state, it matches the outgoing edges reasonably well. States with much coverage to be found, tend to have more outgoing edges; the only exception is state 2.

This leads us to believe that OE might have value that cannot be seen on simple targets where target states get exhausted of coverage to be found. That being said, the difficulty of finding new coverage in a state increases the more time is spent fuzzing that state. For this reason, a static distribution can only get you so far. An adaptive strategy, such as novelty search should have better chances, but this approach does not seem to work due to the sparsity

Config	state distr. (%)	Cov (%)
mcmm-no-1000	(12/20/16/24/28)	36.6
mcmm-oe-1000	(12/20/16/24/28)	35.2
mcmm-ns-1000	(20/20/20/20/20)	39.3
mcmm-cy-1000	(20/20/20/20/20)	36.6

Figure 8: Results experiment 3, showing the distribution of the selected target states, the coverage percentages and the test case throughput in test cases per second.

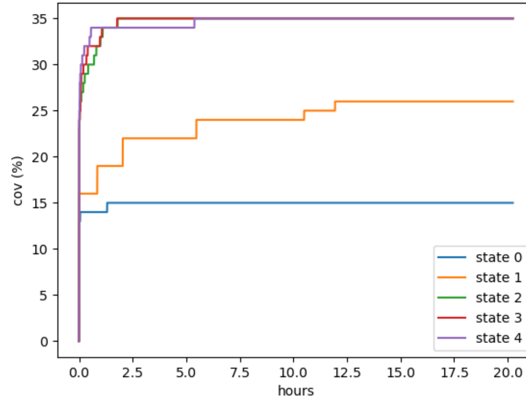

Figure 9: Coverage over time per state from mcmm-ns-1000 (effectively cy)

of finding new coverage. To confirm these suspicions, experiments with other, more complex targets are needed.

From these results we can conclude that this implementation of novelty search does not improve the fuzzer's ability to find code coverage. Although OE does not outperform CY in with a target such as LightFTP, the fact that the distribution roughly matches the amount of code coverage that can be found per state suggests that OE might outperform CY on other targets.

In conclusion: To answer question (D), the novelty search algorithm has no merit with it's current approach because it is barely used during fuzzing process. The state scheduling algorithm CY seems to slightly outperform the OE algorithm, although, the data is not definitive. Moreover, there is reason to suspect that the OE algorithm might be better than CY when fuzzing other targets.

7 Limitations

There are several limitations to both the approach and the evaluation. First, we will discuss the limitations of the approach and afterwards the limitations of the evaluation and the validity of the results.

7.1 Disadvantages of having multiple corpora (Approach)

While having state-aware test cases, implemented as having a separate corpus for each state in Fallaway, makes the feedback sounder (states no longer affect each other), it also has a downside.

Any shared behavior between target states needs to be discovered for each target state

separately. Especially if this shared behavior is difficult to find, this is a waste of time. Speed is important in fuzzers and we should try to avoid doing the same work more than once.

One of the requirements stated in subsection 3.2 is that states do not negatively influence each other due to sharing information such as test cases. Our approach limits the influence states have on each other, fulfilling the requirement that states do not negatively influence each other, but neglects that states might very well positively influence each other in the case of shared behavior. Sharing test cases can be an asset as well as a detriment.

A possible solution to this problem is to share interesting test cases. If a test case is found that is interesting, i.e., it found new code-coverage, we also execute the test case in all other states. In case of shared behavior, the test case would be found interesting for both target states.

7.2 Disadvantages of target state reuse (Approach)

There are clear benefits to the target state reuse, as was seen in the evaluation. However, there are some disadvantages to this approach. As noted in subsection 4.4, whenever a state transition is triggered, subsequent feedback will be attributed to the wrong target state.

Moreover, test cases in our corpus might cause us to prefer certain paths through the target state machine, causing us to spend less time in certain states than we would like. For example, imagine that there is a test case, $t_{A \to B}$, in the corpus that causes a state transition from state A to state B. Whenever we are in state A and the seed scheduler selects $t_{A \to B}$, $t_{A \to B}$ is mutated into a slightly different test case, t', which is executed. There is a high chance t' still causes the same state transition. This is problematic, because the more of these kinds of test cases we have in our corpus, the less time we are likely to spend in state A. So, as we discover more test cases that cause state transitions, we will spend less time in state A because the likelihood of transitioning goes up. This is also true for any state. It is possible that eventually test cases that cause state transitions cause the fuzzer to funnel to deep states, causing the fuzzer to spend little time in shallower states. However, since we do not keep track of what state the target is in, we cannot verify this.

These problems could be alleviated if we were able to track the state the target is in, something we considered out of scope for this research. If we can detect state transitions, we can update the currently selected state in the fuzzer to match the internal state of the target whenever there is a state transition. Then, feedback and test cases will be attributed to the correct target state in the fuzzer.

Moreover, state feedback allows us to monitor if a test cases in our corpus has a high likelihood of causing a state transition and take action accordingly, such as removing it from the corpus.

7.3 Crash triaging (Approach)

A problem that exists in AFL* and is inherited by Fallaway is the increased difficulty of crash triaging. By crash triaging we mean the process of taking the crashing input from the fuzzer and reproducing and understanding why the crash happens.

In AFLnet, whenever a crash is found, the whole trace is collected, which contains all information to reproduce the crash. Doing the same in AFL* and Fallaway is problematic, because the number of inputs sent before the crashing input can be very large. If we reuse the target state many times, i.e., we choose a high m (see subsection 2.3), we have to save m inputs to reproduce the crash. The current implementation of Fallaway simply saves every single input tried, such that crashes will always be reproducible. However, this causes high disk space usage as well as slightly slowing down the fuzzer due to many writes to the disk.

An improvement over the current implementation would be to only save the inputs which either cause a crash or those that find some new code-coverage. With this approach, most inputs can be discarded.

Regardless, triaging is more difficult than with AFLnet when a crash is found. Most of the inputs do not affect the crash and are effectively noise, every input *could* be vital because they cause a state transition. Then, the crash might not occur without also sending this input.

A possible improvement would be to create a minimizer utility, much like AFL's `afl-tmin`, for those familiar. `afl-tmin` is a utility provided by AFL that minimizes test cases to the smallest size such that it still causes a crash and still has the same code-coverage. A similar approach could be taken to minimize the number of the crashing inputs from Fallaway, by automatically trying to leave out inputs and seeing if the target still crashes and the code-coverage does not change.

7.4 Difficulties with patching the source code to enable AFL persistent mode fuzzing (Approach)

To enable fuzzing a target with Fallaway, the target's source code needs to be patched to enable AFL persistent mode fuzzing. With respect to Fallaway, the goal of persistent mode is to patch the target such that it stops its own execution just after it has processed an input.

For some targets, such as LightFTP, this is easy. For LightFTP, there is a single send-receive loop that processes an input and sends back a response.

However, for other targets, this might not be trivial. One example: When attempting to patch OpenVPN, we ran into the issue that the program is event driven. The program has an event loop that processes events, such as, data can be read on the socket or tap/tun device, data can be sent on the socket tap/tun device, the next packet fully formed. Therefore, extra logic is needed to make sure the correct events are processed before letting the target stop itself, but not too late such that it blocks itself waiting for more input. This stranded our effort and requires expert knowledge to properly patch.

Another example: When trying to patch OpenSSH, the problem arose that there is no loop that is responsible for processing all input received over the socket. The input that has to do with authentication is processed in a completely different part of the code than input processed when a connection is already established. While not impossible, patching these kinds of software is difficult and requires good knowledge of the source code.

7.5 Small sample size (Evaluation)

The experiments performed are small scale. Fuzzing is an inherently random process, so different runs of the fuzzer over the same target can get quite different results. Ideally, the fuzzer would have been run numerous times for each configuration to average out the randomness. As a result, it is hard to draw definitive conclusions from the results, because a fair bit of randomness is involved. Running each configuration numerous times turned out to be hard due to time constraints.

7.6 Generality of the approach (Evaluation)

The evaluation was performed with only one target program, namely LightFTP. This is a relatively simple program and FTP is also a relatively simple protocol. This makes it difficult to prove the general applicability of the approach. Perhaps Fallaway works wonderfully well for LightFTP, but not for other targets.

No design decisions are tailored towards FTP or LightFTP. However, the motivation to create Fallaway arose because AFL* performed well, but AFL* has only been tested on LightFTP. One reason that AFL* and Fallaway might perform better on a protocol like FTP and specifically LightFTP is because it is rather forgiving. When an input cannot be parsed, it is simply ignored. If a protocol transitions to the initial state or an error state whenever a malformed input is processed, reusing the target state might be hurtful to the performance of the fuzzer.

Such problems could be resolved by patching the source code of the target to ignore these inputs instead of transitioning to an error state.

Regardless, testing Fallaway on another target would have strengthened the argument that Fallaway is a general approach.

7.7 Is AFLnet State of the Art? (Evaluation)

The argument could be made that AFLnet is no longer the state of the art. Some fuzzers have been proposed that have also outperformed AFLnet, such as SGFuzz [3] and NSFuzz [24]. It might have been better to compare Fallaway to one of these fuzzers.

NSFuzz is not publicly available. SGFuzz is publicly available, but is not integrated with ProFuzzBench, making it time consuming prepare it to fuzz to LightFTP.

8 Related work

There has been a lot of research into fuzzers for stateful systems. After AFLnet was proposed – the first coverage-based stateful fuzzer – most stateful fuzzers proposed have also been coverage-based. That being said, different approach have also been proposed, such as man-in-the-middle fuzzers, those focussing on state model inference and machine learning-based fuzzers.

8.1 Code coverage-based stateful fuzzers

The following fuzzers for stateful systems have different kinds of state feedback mechanisms and are all code coverage-based.

AFLnet [23] is one of the first coverage-guided *stateful* fuzzers. It modified AFL to use network sockets as a test case input, enabling it to fuzz protocol implementations. Instead of messages, it works on traces. It first sends messages to get the SUT in a particular state. Then it sends a mutated messages to explore said state.

Apart from the mutation capabilities inherited from AFL, AFLnet uses the responses from the SUT to build state machine model. It attempts to capture what state the SUT is in, based upon its responses. This is used as extra feedback to guide AFL in choosing seeds.

AFLnet is used as the base of other fuzzers [3, 21, 24] and it is often used to compare the performance of newly proposed fuzzers to [3, 15, 19, 21, 24].

StateAFL [21] builds on AFLnet. It improves the feedback mechanism that infers what state the SUT is in during fuzzing. StateAFL monitors long-lived memory regions which it groups into states by using a fuzzy hash algorithm. StateAFL creates more accurate state models than AFLnet but due to heavy instrumentation, the eventual increase in code coverage is not large.

SGFuzz [3] statically analyses the code in advance to find and instrument variables which it assumes are used to capture the state of the SUT.

NSFuzz [24] synchronized the SUT and the fuzzer to eliminate waiting times by finding the send-receive loop through static analysis. It also refined the method SGFuzz uses, by eliminating variables that are not updated within the send-receive loop of the SUT.

8.2 Man-in-the-Middle Fuzzers

These fuzzers take a different approach. Man-in-the-middle fuzzers position themselves in between the client and the server to intercept and mutate messages. This approach relieves the fuzzer from having to generate valid messages themselves. AutoFuzz [14] is such a fuzzer. It first passively inspects the traffic, without mutating anything, to approximate the state machine. Afterwards it starts fuzzing the server by mutating intercepted messages. When doing this it uses its knowledge of the state machine to guide the fuzzing.

Another man-in-the-middle fuzzer is SECfuzz [26] which tries to deal with encryption. It requires a dump of the cryptographic parameters and key and a protocol specific mapper that can take this information to perform the decryption and re-encryption.

Bleem [19] is a black-box protocol fuzzer that focuses on fuzzing the packet sequence, as opposed to individual packets. It views the server and client as one system and intercepts, mutates and reorders packets using a proxy. It utilizes the client and server to help instantiate packets, instead of reimplementing it. Bleem also supports packet-level mutations but focusses on sequence-level mutations, namely packet duplication and disordering.

8.3 State Model Inference and Analysis

Another approach to test protocol implementations is extracting the finite state machine (FSM) that describes the system under test (SUT) and checking it for correctness. This means that only bugs that manifest in the state machine can be found. Extracting the FSM is done with a technique called regular inference, either by passively learning from data, or actively by querying the SUT. Active learning uses the L* algorithm [1] or the improved TTT algorithm [17], all implemented in LearnLib [25]. The algorithm iteratively improves the FSM by querying the SUT. When the FSM could be correct, the algorithm checks whether the FSM corresponds to the SUT by attempting to find a counter example, up until a certain depth. If a counter example is found, the FSM is further refined, otherwise it is accepted. Afterwards the FSM can be inspected for strange states and edges.

De Ruiter et al. [7] used active learning with LearnLib to infer FSMs of TLS implementations. For regular inference to work, an abstract input and output alphabet needs to be known. To achieve this for the TLS implementations, they constructed a mapper between concrete and abstract messages to create a bridge between LearnLib and the SUT. Afterwards, the FSMs were carefully analyzed by hand. As the authors noted, in the case of security protocols, the paths towards a successful connection enabling the exchange of application data are of interest. This greatly decreases the number of paths in the FSM that need to be analyzed.

A few years later, Daniel et al. [5] used a similar approach to analyze different implementations of OpenVPN and OpenVPN-NL, a stripped and hardened version of OpenVPN.

Inspired by De Ruiter et al., Friterau-Brostean et al. [11] used largely the same technique to analyze DTLS implementations. They again used LearnLib and afterwards analyzed them by hand. DTLS is more difficult to prepare for LearnLib because it is complex. Since it runs over UDP, it has to reimplement a reliable channel where TLS relies on TCP.

These papers discovered problems in and interesting aspects of the implementations of these protocols, such as unintended edges and superfluous states. Using this kind of state model inference as state feedback might be interesting. However, while useful, all approaches were tailor made to a protocol and required expertise and manuel effort to execute.

8.4 Machine Learning Fuzzers

Another type of fuzzers are those that use machine learning [8, 16, 28]. These fuzzers train a machine learning model on a set of traces of network traffic. This model is then used to generate traces that are *almost* correct, which is exactly what is needed for effective fuzzing.

One limitation of this approach, as with all approaches that learn from captured traces, is that the test case generation can only be as diverse as the set of traces it is trained on. If some functionality is not found in the training data, these fuzzers are unlikely to generate test cases with this behavior.

9 Conclusions

We set out to improve AFL* by proposing a new approach that extends AFL* to be state-aware. This new approach combines AFL* with parts from AFLnet. AFL* has high test case throughput due to reuse of the target process, but does not have awareness of states. On the other hand, AFLnet is state-aware, but very slow. Our approach, called Fallaway, combines these approaches and is implemented using LibAFL, a library to build modular fuzzers written in Rust.

Our evaluation shows that Fallaway outperforms AFLnet when fuzzing LightFTP in terms of code coverage. This performance increase seems to be caused solely by the reuse of the target process. The state-awareness of the test cases and feedback do not seem to affect the performance significantly.

There are limitations to Fallaway's current approach. Solving these problems, such as sharing interesting test cases between states and incorporating state feedback, thereby partially negating the disadadvantages of target state reuse could improve the performance of Fallaway.

Target state reuse seems to be a worthwhile technique to use. Moreover, using LibAFL for future research would be great useful to ease the comparison of approaches in the future.

10 Future Work

First, the limitations of the approach should be addressed. Avoiding having to discover the shared behavior more than once seems straightforward by executing test cases that found code coverage on each other target state. Moreover, extending Fallaway with state feedback such that the state of the target can be approximated might help with the disadvantages of the target state reuse. Also, improving the crash triaging process will increase the usability of the approach.

Secondly, testing Fallaway on different targets will shed more definitive light on the generality of the approach. Targets that are inside of ProFuzzBench are good candidates for targets, because the targets are already prepped. For example, randomness is often already removed.

Lastly, we have gained appreciation of LibAFL to build fuzzers and think it could be extended for stateful systems. The purpose of Fallaway was not to be perfectly modular, but due to the nature of LibAFL, there is quite some modularity to it. To make it truly modular an extension of LibAFL with components specifically for stateful targets seems feasible, a LibStateAFL if you will. The API would have to be polished, but Fallaway already set the first steps both in terms of approach and implementation.

It would be a great enabler of future research into fuzzers for stateful systems, since it allows comparisons where only the new approach is changed, and the rest stays the same.

The main components that need to be supplied are resettable targets, a fuzzer that knows about states, prefixes and a state scheduler, as well as a state object that knows how to deal with this. Other fuzzers prefer to use full traces as test cases instead of single messages. This could also be provided.

References

[1] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106.

[2] Anonymous. "AFL*: A Simple Approach to Fuzzing Stateful Systems". anonymous preprint under review. 2024. URL: https://openreview.net/forum?id=gmXkDLjole.

[3] Jinsheng Ba et al. "Stateful greybox fuzzing". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3255–3272.

[4] Osbert Bastani et al. "Synthesizing program input grammars". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 95–110.

[5] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. "Inferring OpenVPN state machines using protocol state fuzzing". In: *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2018, pp. 11–19.

[6] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. *Fuzzers for stateful systems: Survey and Research Directions*. 2023. arXiv: 2301.02490 [cs.CR].

[7] Joeri De Ruiter and Erik Poll. "Protocol state fuzzing of TLS implementations". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 193–206.

[8] Rong Fan and Yaoyao Chang. "Machine learning for black-box fuzzing of network protocols". In: *Information and Communications Security: 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings 19*. Springer. 2018, pp. 621–632.

[9] Andrea Fioraldi et al. "AFL++: Combining incremental steps of fuzzing research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

[10] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*. CCS '22. Los Angeles, U.S.A.: ACM, Nov. 2022.

[11] Paul Fiterau-Brostean et al. "Analysis of DTLS implementations using protocol state fuzzing". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2523–2540.

[12] *GitHub - gcovr/gcovr: generate code coverage reports with gcc/gcov — github.com*. https://github.com/gcovr/gcovr. [Accessed 07-05-2024].

[13] *GitHub - hfiref0x/LightFTP: Small x86-32/x64 FTP Server — github.com*. https://github.com/hfiref0x/LightFTP. [Accessed 06-04-2024].

[14] Serge Gorbunov and Arnold Rosenbloom. "Autofuzz: Automated network protocol fuzzing framework". In: *Ijcsns* 10.8 (2010), p. 239.

[15] Jiaxing Guo et al. "Stateful Black-Box Fuzzing for Encryption Protocols and its Application in Ipsec". In: *Available at SSRN 4563904* (2023).

[16] Zhicheng Hu et al. "GANFuzz: a GAN-based industrial network protocol fuzzing framework". In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 2018, pp. 138–145.

[17] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT algorithm: a redundancy-free approach to active automata learning". In: *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer. 2014, pp. 307–322.

[18] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018), pp. 1–13.

[19] Zhengxiong Luo et al. "BLEEM: Packet Sequence Oriented Fuzzing for Protocol Implementations". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 4481–4498.

[20] Valentin JM Manès et al. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331.

[21] Roberto Natella. "Stateafl: Greybox fuzzing for stateful network servers". In: *Empirical Software Engineering* 27.7 (2022), p. 191.

[22] Roberto Natella and Van-Thuan Pham. “Profuzzbench: A benchmark for stateful protocol fuzzing”. In: *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 2021, pp. 662–665.

[23] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. “AFLNet: A Greybox Fuzzer for Network Protocols”. In: *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*. 2020.

[24] Shisong Qin et al. “NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing”. In: *ACM Transactions on Software Engineering and Methodology* (2023).

[25] Harald Raffelt et al. “LearnLib: a framework for extrapolating behavioral models”. In: *International journal on software tools for technology transfer* 11 (2009), pp. 393–407.

[26] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. “SECFUZZ: Fuzz-testing security protocols”. In: *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE. 2012, pp. 1–7.

[27] Michal Zalewski. *American fuzzy lop (AFL) fuzzer*. URL: https://lcamtuf.coredump.cx/afl/.

[28] Hui Zhao et al. “SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective”. In: *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE. 2019, pp. 59–67.