UNIVERSITÀ DEGLI STUDI DI CATANIA
Dipartimento di Matematica e Informatica
Corso di Laurea Triennale in Informatica

Leonardo Cantarella

Benchmarking Stateful Fuzzers over Lighttpd

Final Project Report

Supervisior: Giampaolo Bella
Advisor: Marcello Maugeri
External Advisor: Cristian Daniele

Academic Year 2023 - 2024

Abstract

Fuzzing is a software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. This technique is widely used to identify vulnerabilities in software systems.

The significance of stateful fuzzing lies in its ability to identify vulnerabilities in applications characterized by intricate internal states, which may be overlooked by conventional fuzzing techniques.

This thesis compare three stateful fuzzers—Fallaway, AFLNet and ChatAFL—by employing them against Lighttpd, a high-performance web server. This research compares these instruments with regard to code coverage, crash detection and the different kinds of eventually vulnerabilities.

These results provide enlightening insights into the strengths and weaknesses of each fuzzer, hence guiding selection and improvements of stateful fuzzing approaches for modern software systems.

Contents

1 Introduction

As the software systems are getting increasingly complex, ensuring their robustness and security has turned out to be a serious challenge. In this scenario, fuzzing has emerged as a powerful technique in the identification of security vulnerabilities and defects, which may remain elusive for traditional testing techniques. Fuzzing involves the generation of random test inputs in order to see how the software reacts to unexpected or malformed input data, looking for problems, such as crashes, unexpected behaviour, or security vulnerabilities.

The evolution of fuzzing methodologies has significantly enhanced their effectiveness for a wide range of applications. Traditional fuzzers typically focus on generating random inputs and observing the software responses. However, for applications that maintain internal states across several interactions, such as web servers or networked applications, this approach can be insufficient. These stateful applications call for more sophisticated fuzzing techniques that take into account the interaction between different states and transitions.

Stateful fuzzing is an advanced approach for solving the problems of applications that rely on state management. Whereas in stateless fuzzing, each input is considered a unique event, stateful fuzzing emulates the flow of activities along with the succeeding changes in the state of an application. It includes the generation of inputs which consider previous interactions and what these have done to the state of the application, hence providing a more realistic and deeper testing process.

An instance used in this thesis is Lighttpd, an open-source web server recognized for its effectiveness and ability to scale. Lighttpd is engineered to manage a substantial number of concurrent connections and accommodates multiple network protocols. Assessing Lighttpd offers a chance to scrutinize stateful fuzzing methodologies.

This thesis focuses on the benchmarking of stateful fuzzers to ascertain their efficiency in coverage and detecting eventually vulnerabilities in Lighttpd. Our evaluation stateful fuzzers are: Fallaway, AFLNet and ChatAFL. Each of them has a its own approach toward stateful fuzzing. By analyzing the performance of all these tools, we aim to report on various strengths and weaknesses of each, which gives necessary suggestions for improving stateful fuzzing techniques and enhancing the security of modern software systems. This introductory section creates a base for deep research in stateful fuzzing methodologies and their realization in real systems, setting the base for further chapters examining tests and results.

2 Background

2.1 Introduction to Fuzzing

Fuzzing, or fuzz testing, is a software testing technique that includes feeding a huge amount of random data into the system, called SUT (System Under Test), to find unprecedented responses and reveal major programming errors, along with key security vulnerabilities. The primary objective of fuzzing is to identify vulnerabilities such as buffer overflows, memory leaks and other security weaknesses that can be exploited by attackers.
The success of fuzzing is based on its capabilities for automatic test case generation and for focusing its attention on portions of programs that otherwise would not have been tested by other more traditional testing technique.
Indeed, it is particularly effective for applications with complex input grammars, where manual test case creation would be impracticable.

2.1.1 Types of Fuzzing Techniques

Fuzzing methodologies vary and there exist a lot for different applications and purposes:

- **Black-box Fuzzing**: This is a technique of generating inputs without prior knowledge of the internal structure of an application. It is easy to deploy but often less efficient as there is no internal feedback.

- **White-box Fuzzing**: This is one of those techniques that rely heavily on source code intuition, such as control flow and data flow, to provide maximum code coverage with test case generation. The approach often employs some sort of complex static and dynamic analysis methodologies.

- **Grey-box Fuzzing**: It's a strategy that combines the various merits of black-box and white-box fuzzing. It brings in partial knowledge about internal application details and code coverage feedback guiding the generation of inputs. It balances simplicity with effectiveness and an example might be the family of tools called AFL (American Fuzzy Lop).

2.1.2 Fuzzing Inputs Generation

There are also different ways to generate inputs for fuzzing:

- **Mutation-based Fuzzing**: This generates new inputs through random mutations of existing inputs (for example modifying bits or bytes of existing test cases). It requires no knowledge about the structure of the inputs but is often a lot weaker compared with other generations for applications requiring highly structured inputs.
- **Generation-based Fuzzing**: This builds the inputs from scratch, based on a formal characterization of the input format, grammar or protocol specification. It has proved quite effective in applications where the inputs have to be complex or systematically structured.

2.1.3 Coverage-Guided Fuzzing

Coverage-guided fuzzing (CGF) is a subtype of grey-box fuzzing that leverages code coverage information to drive the generation of test inputs. It aims to explore as many code paths as possible by continuously generating inputs that maximize the coverage.

For example, the American Fuzzy Lop (AFL) fuzzer is a popular coverage-guided fuzzer that uses a feedback loop to guide the generation of new test cases. AFL instruments the binary to track the code coverage during execution and uses this information to guide the mutation of test cases. The fuzzer maintains a queue of test cases and iteratively selects, mutates and executes them to maximize the code coverage.

In this context is also important to describe the concept of **edge coverage**, that is a metric that measures the number of unique edges traversed by the program during execution. An edge is a transition between two basic blocks in the control flow graph of the program (a basic block is a sequence of instructions not containing any jumps or branches). For example, consider the following code snippet:

```
if (x > 0) {
    y = 1;
} else {
    y = 2;
}
```

In this case, there are two edges (Figure 2.1): one from the condition to the true branch and one from the condition to the false branch. The basic blocks are the condition, the true branch and the false branch.
This edges are used in the **coverage map**, that is a binary compiled with edge coverage information, where each edge is mapped to a bit in the coverage map.

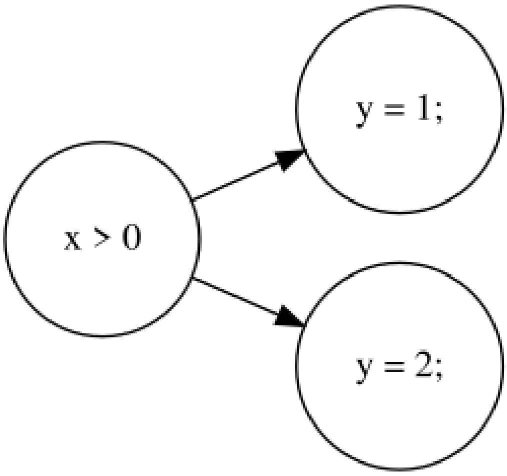

Figure 2.1: Example of edge between basic blocks

When an edge is executed, the corresponding bit in the coverage map is set to a probability value. The fuzzer uses this information to guide the generation of new test cases that maximize the coverage.
Within this action a coverage-guided fuzzer maintains a collection of inputs called **corpus**. In particular it is a collection of:

- **Seeds**: Initial inputs that are used to start the fuzzing process.
- **Generated inputs**: Inputs that are generated by the fuzzer during the fuzzing process (i.e. by mutating the seeds).

The corpus grows as the fuzzer adds new inputs that has allowed it to increase the coverage of the program. To choose which input to mutate next, the fuzzer has a **queue**, that is a scheduling algorithm that selects the next input to use based on different strategies like:

- **Coverage-Yield (CY)** strategy, which uses a round-robin approach to select the next input to mutate.

- **Outgoing Edges (OE)** strategy, which prioritizes inputs that exercise new edges in the program (this is done by using metadata that keeps track of the outgoing edges of each state in the program and selecting the input that exercises the most new edges).

Corpus and queue are used interchangeably, but they are not really the same thing. We can look at the queue as a mapping of the corpus within a scheduling algorithm to choose the next input to mutate. The mix of corpus, coverage map and queue is the core of the coverage-guided fuzzing strategy.
These are some examples of coverage-guided strategies:

- **Multiple Corpus Single Map (MCSM)**: This strategy uses a single coverage map to track the coverage of the program and multiple queues to store the inputs (is more efficient in terms of memory usage and faster in terms of execution time, but it is less effective in terms of coverage).
- **Multiple Corpus Multiple Map (MCMM)**: This strategy uses multiple coverage maps to track the coverage of the program and multiple queues to store the inputs (is more effective in terms of coverage, but it is less efficient in terms of memory usage and slower in terms of execution time).

Each queue contains inputs that have allowed the fuzzer to reach a specific state of the program. The fuzzer uses a scheduler to select which input to mutate next based on the coverage achieved by the input.

2.2 Stateful Fuzzing: Concepts and Challenges

Stateless fuzzing is a traditional fuzzing technique that generates random inputs to test the behavior of an application. However, this approach is not always effective for applications that maintain internal states across multiple interactions.
For example cosidering an FTP server, like LightFTP, until the user is not authenticated, all the inputs are meaningless. In this case, the fuzzer should be able to generate a sequence of inputs that first authenticate the user and then test the behavior of the application. *Stateful fuzzing* adds state awareness to traditional fuzzing methods. It considers an application's internal state and how that state might affect subsequent inputs handling. This becomes particularly critical for applications that handle complex state information, such as network servers, databases and interactive applications.

2.2.1 Understanding Stateful Applications

Stateful applications maintain state across multiple interactions or sessions. Examples include network servers that manage connection states, authentication states, session identifiers, or other state information specific to an application. These states significantly influence the processing of inputs and the behavior of the application over time.
Good practices in state transition management are crucial for both security and reliability: bugs related to state transitions can lead to vulnerabilities such as unauthorized access, denial of service (DoS), or data corruption.
Stateful fuzzers attempt to model and explore these state transitions by generating input sequences that mimic valid usage scenarios while concurrently monitoring state changes to ensure comprehensive coverage of all possible transitions. To better understand this concept, consider a simple state model shown in Figure 2.2.

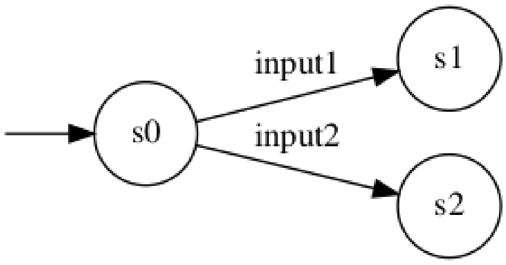

Figure 2.2: A simple state model illustrating state transitions in a stateful application

2.2.2 Key Techniques in Stateful Fuzzing

Stateful fuzzing involves several advanced techniques that distinguish it from traditional fuzzing approaches:

- **State Modeling**: The process of building a model representing an application state machine by reverse engineering source code, observing real interactions, or guide test case generation in conjunction with machine learning techniques.
- **State Tracking**: This involves tracking the state of the application across successive inputs; indeed, tracking of network traffic, system calls, or internal state variables.

- **Feedback Mechanisms**: With feedback mechanisms, one can prioritize those test cases that tend to explore new states or code paths; hence, the general efficiency of fuzzing can be improved.

- **Sequence Generation**: It is the need to generate input sequences to properly model actual use, since the findings of vulnerabilities often depend on specific sequences or state transitions.

- **Learning-Based Approaches**: Certain fuzzers utilize machine learning or heuristic methodologies to dynamically ascertain the structural configuration of the application's state machine, thereby enabling the fuzzer to adjust and enhance its efficacy progressively.

2.2.3 Challenges in Stateful Fuzzing

Successfully performing testing is fraught with several challenges in stateful fuzzing:

- **State Explosion**: As in real life, an application itself may have a number of possible states and with more states, a risk for exponential growth in process complexity increases. In this case, state abstraction, pruning, or prioritization counters the *state explosion* in an essential way.

- **Protocol Complexity**: Generating meaningful input sequences can involve deep knowledge of complex protocols or state machines. This often includes much domain-specific knowledge or even advanced algorithms.

- **Performance Overhead**: To date, state tracking performed by the application and input sequence generation can cause significant computational costs, hence slowing down the fuzzing process.

- **Handling Non-Deterministic Behavior**: The nondeterministic behavior of stateful applications often results from concurrency, differences in external inputs, or even timing variations. These factors therefore make the reproduction of bugs and receiving consistent fuzzing results usually difficult.

2.3 Lighttpd: A Case Study for Stateful Fuzzing

Lighttpd is an open-source web server optimized for performance with very low memory usage. It is designed to handle huge volumes of parallel connections with minimal overhead, making it particularly useful on systems with limited resources or those requiring a high degree of concurrency. Its modular design and support for advanced web protocols make it a popular choice for embedded systems, cloud computing platforms and high-traffic websites. It was used by popular websites like Wikimedia, YouTube and SourceForge, nowadays is used in general for ...

2.3.1 Overview of Lighttpd Architecture

Lighttpd operates on an event-driven architecture, which enables it to serve many requests concurrently. An asynchronous I/O framework is employed to minimize overhead in network connections, allowing the server to scale efficiently under varying workloads. The key features of Lighttpd include:

- **Modular Design**: Provides a series of modules for implementing functions like URL rewriting, HTTP compression, SSL/TLS and WebSockets. The modular design allows for customization based on specific needs.

- **Protocol Support**: Out of the box, it supports HTTP/1.1, HTTPS, FastCGI, SCGI and HTTP/2, making it suitable for a wide range of web applications and services.

- **Security Attributes**: Advanced integrated security features include TLS/SSL encryption, prevention of denial-of-service attacks and multiple authentication options.

2.3.2 Relevance of Lighttpd for Fuzzing

Lighttpd is an important SUT for fuzzing due to its common use behind various internet applications. These characteristics make it a suitable candidate for evaluating fuzzing techniques. By default, Lighttpd is not stateful, but it maintains transient states during the processing of requests (Figure 2.3).

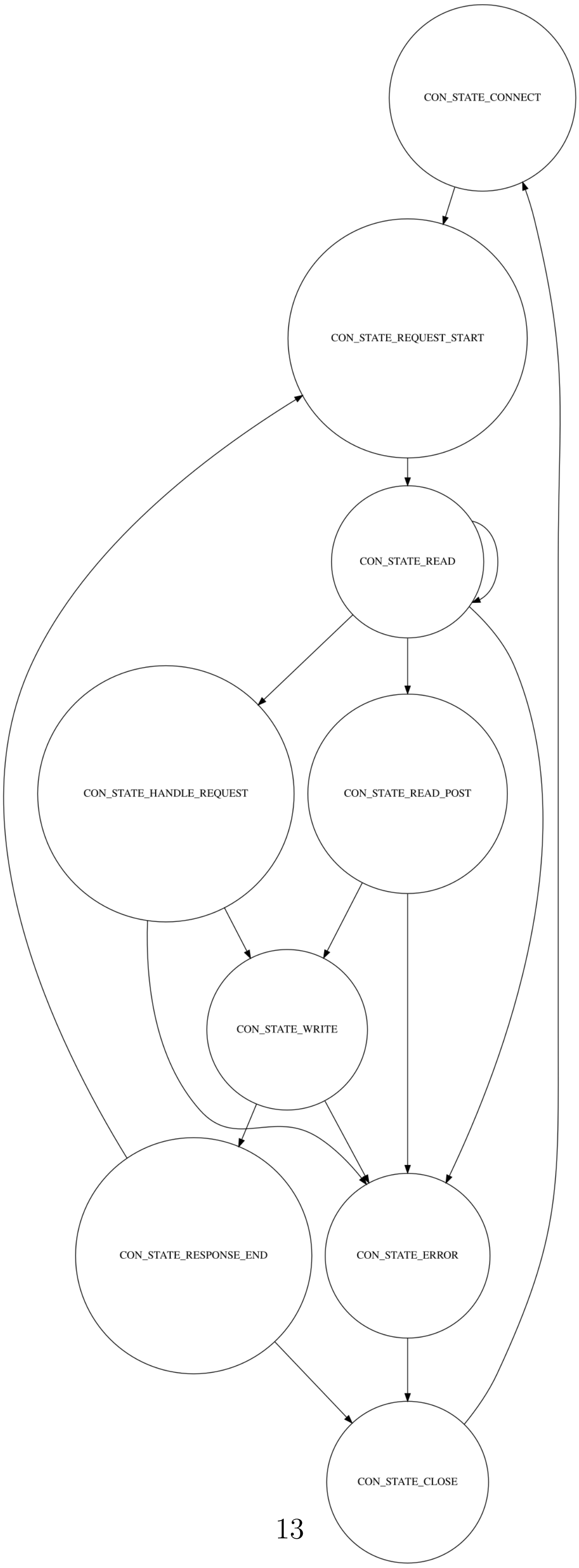

Figure 2.3: State model of Lighttpd

The state model seems to be complex, but the state are effectively transitory. A sample flow of states in Lighttpd is as follows:

- **CON_STATE_CONNECT**: The initial default state before a connection is established.
- **CON_STATE_REQUEST_START**: The state right after a connection is established and wait for request.
- **CON_STATE_READ**: The state when the server is reading the request (here the server can be in a loop to read the request).
- **CON_STATE_HANDLE_REQUEST**: The state when the server is processing the request, if body length is null.
- **CON_STATE_READ_POST**: The state when the server is processing the request, if body length is more than 0.
- **CON_STATE_WRITE**: The state when the server is writing the response.
- **CON_STATE_REQUEST_END**: The state after the request has been fully received.
- **CON_STATE_CLOSE**: The state when the connection is closed.

In the *CON_STATE_CONNECT* we have no control of input, because it only changes when the connection is established.
The only stationary state could be the *CON_STATE_READ*, because it is a loop that reads the request (i.e. if we send line by line the request, we will loop into it).
The other states are transitory because, at the end, we will go back to *CON_STATE_REQUEST_START* or *CON_STATE_CONNECT*.
For this thesis, we have chosen to model the state of the server based on the existence or non-existence of a resources (Figure 2.4). A resource can be a file, a directory, or any other entity that can be accessed via HTTP. By considering two types of requests—those that attempt to access an existing resource and those that attempt to access a non-existing resource—we can effectively explore different states of the server.

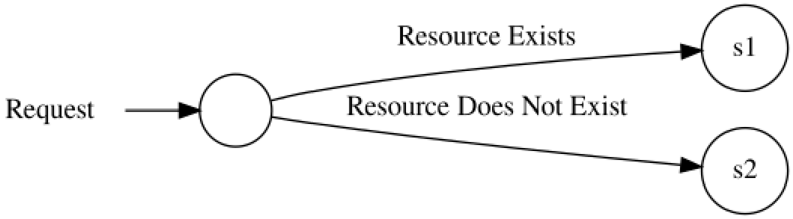

Figure 2.4: State model of Lighttpd based on the existence of resources

2.4 Fuzzers Overview: AFLNet, ChatAFL and Fallaway

For this thesis, three stateful fuzzers—AFLNet, ChatAFL and Fallaway—will be benchmarked over Lighttpd.

2.4.1 AFLNet

AFLNet [1] is a stateful CGF tool. It integrates automated state model inference with coverage-guided fuzzing, creating a synergistic relationship between the two processes. As fuzzing generates new message sequences to reach unexplored states, it progressively builds a more complete state model. Concurrently, this dynamically evolving state model helps guide fuzzing efforts towards more significant areas of the code, leveraging both state and code coverage information of the retained message sequences.

AFLNet is implemented as an extension of the popular grey-box fuzzer AFL, with the additional capability of facilitating **network communication** over sockets, which is not supported by the original AFL. To achieve this, AFLNet establishes two communication channels: one for sending messages to the SUT and another for receiving responses. The response-receiving channel acts as a state feedback channel, complementing the code coverage feedback channel utilized by other CGF tools. The communication is implemented using standard C Socket APIs and synchronization between AFLNet and the server is ensured by introducing delays between requests.

AFLNet uses a **prefix-based fuzzing** strategy, where the fuzzer maintains a prefix of the message sequence that has been successfully processed by the server. This prefix is used to guide the generation of new message sequences, ensuring that the fuzzer explores new states and code paths while maintaining the validity of the input.

The input to AFLNet consists of pcap files capturing network traffic, such as interactions between a client and server. A network sniffer, like tcpdump, is used to capture realistic exchanges and a packet analyzer, such as Wireshark, can automatically extract the relevant message sequences. AFLNet uses a **Request Sequence Parser** to generate an initial corpus of message sequences by parsing these pcap files. It isolates individual client requests, discards the responses and identifies the beginning and end of each message, utilizing protocol-specific markers.

The **State Machine Learner** component then augments the protocol state machine with newly observed states and transitions by analyzing server responses. AFLNet extracts status codes from server responses to identify and document new states and transitions. The **Target State Selector** leverages this information to determine which state the fuzzer should focus on next. This is done by applying several heuristics based on the statistical data gathered from the state machine, aiming to identify "blind spots" or rarely exercised states and maximize the discovery of new state transitions. Once a target state is selected, the **Sequence Selector** chooses a corresponding message sequence from the corpus that can reach the desired state. AFLNet maintains a state corpus and a hashmap to facilitate efficient selection of sequences. The selected sequence is then subjected to mutation using the **Sequence Mutator**, which builds upon AFL's **'fuzz_one'** method (*input selection, mutation, execution, feedback collecltion, minimization and prioritization, looping*). AFLNet uses protocol-aware mutation operators to modify the candidate subsequence, enhancing the chances of generating new sequences that can lead to the discovery of new states or code branches.

AFLNet employs several mutation strategies, such as replacing, inserting, duplicating, or deleting messages, in addition to standard byte-level operations like bit flipping. Generated sequences deemed "interesting" — those that uncover new states, transitions, or code branches — are added to the corpus for further fuzzing. This evolutionary approach, driven by the continuous enhancement of the message sequence corpus, underpins the effectiveness of AFLNet in achieving comprehensive state and code coverage.

2.4.2 ChatAFL

Traditional *mutation-based protocol fuzzing* relies heavily on recorded message sequences to generate test cases, which can limit its effectiveness in thoroughly exploring the input and state space of complex network protocols. Existing approaches often require detailed, machine-readable protocol specifications, which are labor-intensive to produce and maintain. Furthermore, these approaches may struggle with limited seed diversity and may reach a coverage plateau (no more progress in discovering code paths or states), where further exploration yields diminishing returns.

To address these limitations, recent advancements have explored the potential of Large Language Models (LLMs) to assist in the fuzzing process.
LLMs are a class of neural network models that have demonstrated remarkable capabilities in natural language understanding and generation. They can be fine-tuned for specific tasks and have been successfully applied to a wide range of applications, including language translation, text generation and code completion.
LLMs, such as ChatGPT, are pre-trained on extensive corpora, including publicly available protocol specifications and have demonstrated impressive capabilities in understanding and generating text. This presents an opportunity to leverage LLMs to improve fuzzing strategies by interpreting natural language descriptions of protocols and generating structured, diverse message sequences.
LLM-guided protocol fuzzing uses the capabilities of LLMs to overcome the limitations of traditional mutation-based fuzzers. This method is implemented in *ChatAFL* [2] , a fuzzing tool built upon the AFLNet framework. *ChatAFL* incorporates LLMs to assist in three key areas:

1. **Grammar Extraction:** By querying the LLM, the fuzzer can obtain a machine-readable grammar for the protocol under test. This grammar is used to guide mutations in a way that maintains the structural validity of the messages, thus enhancing the fuzzer's ability to explore new state transitions.

2. **Seed Enrichment:** The LLM is used to diversify the initial seed corpus by generating new message types that are contextually relevant to the protocol. This helps to overcome the limitations posed by a narrow set of initial test cases and increases the likelihood of discovering new protocol behaviors.

3. **Breaking Coverage Plateaus:** When the fuzzer is unable to achieve further state or code coverage, it is considered to be in a coverage plateau. The LLM can be prompted to generate new message sequences aimed at escaping the plateau by triggering unexplored state transitions.

The integration of LLMs into protocol fuzzing offers several benefits:

1. It reduces the dependence on pre-existing machine-readable protocol specifications by leveraging natural language processing capabilities.

2. It enhances the diversity and effectiveness of the fuzzing process by generating a wider variety of input sequences.

3. It aligns with the inherent goals of fuzzing — automation and adaptability — by using LLMs that can be easily guided via prompts to perform specific tasks without extensive reprogramming or manual intervention.

Overall, this LLM-guided approach, as demonstrated in ChatAFL, represents a novel direction in protocol fuzzing, combining traditional techniques with state-of-the-art language models to improve both the breadth and depth of fuzzing campaigns.

2.4.3 Fallaway

Fallaway [3] is a stateful fuzzer designed to address several key challenges faced by traditional fuzzers when handling stateful targets. Unlike stateless fuzzers such as AFL, which send single test cases and expect the target to terminate, Fallaway manages multiple states by incorporating a **dual-loop** structure: an outer loop that selects the target state and an inner loop that sends multiple test cases for the chosen state. This approach helps maintain deliberate focus on specific states, prevents interference between states and ensures that progress in one state does not hinder progress in another.

To achieve these objectives, Fallaway decouples the concepts of state scheduling and test case (seed) scheduling. For each state, a unique prefix is maintained along with a separate corpus of test cases, allowing focused exploration of the target's behavior within that state. Observations and feedback are also stored separately for each state, avoiding the problem of feedback contamination across different states. This strategy enables the fuzzer to maintain a clear distinction between the information gathered in each state, ensuring that the testing process remains unbiased and effective.

Fallaway is built on the LibAFL framework, which is a modular library for developing fuzzers. To make LibAFL suitable for stateful targets, Fallaway extends its functionality in two key ways:

1. It uses AFL's **persistent mode**, designed to keep a target application running continuously between different test cases, rather than starting a new process for each test case. This approach is particularly useful for maintaining and manipulating the application's state across test cases, allowing the target to handle inputs continuously without resetting after each test case, which is crucial for efficient fuzzing of stateful systems.

2. It introduces an outer loop to handle state transitions and reset the target accordingly, ensuring compatibility with LibAFL's existing mechanisms for executing test cases.

By integrating these methods, Fallaway leverages the speed and efficiency of persistent mode fuzzing while maintaining precise control over state transitions. This approach allows it balance execution speed and focus state exploration.

3 Setup and fuzzing

3.1 Fallaway

Fallaway distinguishes itself from other fuzzers like AFLNet or ChatAFL by using a **persistent mode**. This mode allows the fuzzer to maintain the server's state across multiple requests, which is especially useful in scenarios where the server does not reset its state between requests, such as when managing user sessions or maintaining authentication states in a web application.

The persistent mode is implemented by modifying the Lighttpd server to maintain its state between requests. The server operates in a separate process and the fuzzer interacts with it via a socket. The fuzzer sends requests to the server and receives responses, using the results to guide the generation of subsequent requests. This process continues in a loop until the fuzzing session is complete.

To enable this, we must modify the Lighttpd code to ensure that the server continuously receives, processes and responds to requests without shutting down. The changes are made to the function `server_main_loop` in the `src/server.c` file and to the connection handling functions in `src/connections.c` of the Lighttpd source code. The specific code changes are shown in the next section, providing a comparison between the original and modified code.

3.1.1 Lighttpd Code Modifications for Persistent Mode

Table 3.1 presents a comparison of the original and modified code for the `connections.c` file. The modifications to this file are crucial for maintaining an open connection state, ensuring that the fuzzer can interact continuously with the server. We also had to clean all buffers and old data for that connection.

Table 3.2 shows a comparison of the original and modified code for the server.c file. The changes made here are essential for enabling persistent server operation, allowing the fuzzer to manage and maintain server state across multiple requests, looping into the __AFL_LOOP.

Original Code

```
static void connection_handle_shutdown(connection *con) {
    ...
    connection_reset(con);

    /* close the connection */
    if (con->fd >= 0
        && (con->is_ssl_sock
        || 0 == shutdown(con->fd, SHUT_WR))) {
        con->close_timeout_ts = log_monotonic_secs;

        request_st * const r = &con->request;
        connection_set_state(r, CON_STATE_CLOSE);
        if (r->conf.log_state_handling) {
            log_error(r->conf.errh, __FILE__, __LINE__,
                "shutdown_for_fd_%d", con->fd);
        }
    } else {
        connection_close(con);
    }
}
```

Modified Code

```
static void connection_handle_shutdown(connection *con) {
    ...
    connection_reset(con);

    /* keep the connection open and reset it */
    request_reset_ex(&con->request);
    chunkqueue_reset(con->read_queue);
    con->request_count = 0;
    con->is_ssl_sock = 0;
    con->revents_err = 0;
    connection_set_state(&con->request,CON_STATE_REQUEST_START);
}
```

Table 3.1: Comparison of Original and Modified Code for ‘src/connections.c‘

Original Code

```
static void server_main_loop (server * const srv) {
     ...
    server_load_check(srv);

    #ifndef _MSC_VER
    static
    #endif
    connection * const joblist  = log_con_jqueue;
    log_con_jqueue = sentinel;
    server_run_con_queue( joblist ,  sentinel );

    if  ( fdevent_poll(srv->ev, log_con_jqueue != sentinel ? 0 : 1000) > 0)
        last_active_ts  = log_monotonic_secs;
}
```

Modified Code

```
static void server_main_loop (server * const srv) {
     ...
    server_load_check(srv);

    while (__AFL_LOOP(INT64_MAX)) {
        fdevent_poll(srv->ev, -1);

        #ifndef _MSC_VER
        static
        #endif
        connection * const joblist  = log_con_jqueue;
        log_con_jqueue = sentinel;
        server_run_con_queue( joblist ,  sentinel );
    }

    srv_shutdown = 1;
}
```

Table 3.2: Comparison of Original and Modified Code for ‘src/server.c‘

3.1.2 Mutator and Corpus

Another crucial aspect of the fuzzing process involves the corpus and the mutator. In this experiment, we define the state of the server based on the existence 3.1 or non-existence 3.2 of resources. Specifically, we consider two types of requests: one that attempts to access a resource that exists and another that attempts to access a resource that does not exist.

The **corpus** consists of a set of initial test cases that represent these two scenarios. By including requests for both existing and non-existing resources, we ensure that the fuzzer can effectively explore different states of the server.

```
PUT /hello.txt HTTP/1.1
Host: 127.0.0.1:8080
Content-type: text/plain
Content-length: 13

Hello, World!
```

Figure 3.1: Seed of existent resource request

```
DELETE /hello.txt HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.0.1
Accept: */*
```

Figure 3.2: Seed of non-existent resource request

The **mutator**, which is responsible for generating variations of the requests, is relatively straightforward. Its primary function is to modify the existing requests by appending a sequence of characters (’$\backslash r\backslash n\backslash r\backslash n$’) to the end of each request. This modification is essential as it guarantees that the requests are well-formed and adheres to the HTTP protocol standards. By ensuring the requests are properly formatted, we enable the server to parse and process them correctly, which is vital for accurate fuzz testing.

3.1.3 Setting Up the Fuzzing Environment

To run the fuzzer, we need to build a Docker container that includes all the necessary dependencies and the modified Lighttpd server. The Dockerfile below shows the steps to set up this environment.

```
FROM libaflstar

WORKDIR /

# Copy the patch file
COPY ./lighttpd.patch /lighttpd.patch

ENV DEBIAN_FRONTEND=noninteractive

# Install lighttpd dependencies
RUN apt-get install -y \
    autoconf \
    automake \
    libtool \
    m4 \
    pkg-config \
    libpcre2-dev \
    zlib1g-dev \
    zlib1g \
    openssl \
    libssl-dev \
    scons

# Create the root directory for the server
RUN chmod 777 /tmp

# Install

# Set up environment variables for ASAN
ENV ASAN_OPTIONS=’abort_on_error=1:symbolize=0:detect_leaks=0:
   detect_stack_use_after_return=1:detect_container_overflow=0:
   poison_array_cookie=0:malloc_fill_byte=0:max_malloc_fill_size
   =16777216’

# Download lighttpd
ENV CC=afl-cc
ENV CXX=afl-cc
RUN git clone https://git.lighttpd.net/lighttpd/lighttpd1.4.git
   lighttpd
WORKDIR /lighttpd
RUN git checkout 9f38b63cae3e2
RUN git apply /lighttpd.patch
RUN ./autogen.sh
RUN scons CC=/AFLplusplus/afl-cc CXX=/AFLplusplus/afl-cc -j 4
   build_static=1 build_dynamic=0
RUN mv /lighttpd/sconsbuild/static/build/lighttpd /lighttpd/
   lighttpd

# Copy the corpus
```

```
COPY ./corpus /corpus

# Copy the config file
COPY ./lighttpd.conf /lighttpd.conf

# Copy the run script
COPY ./run.sh /LibAFLstar/run.sh
# Make it executable
RUN chmod +x /LibAFLstar/run.sh

WORKDIR /LibAFLstar
```

The Docker container is configured with all the dependencies to run the fuzzer and build the modified Lighttpd server, providing a controlled environment to conduct the fuzzing experiment. Another important file to consider is the configuration file of the Lighttpd server, which is shown below.

```
    server.document-root = "/tmp"
    server.bind = "0.0.0.0"
    server.port = 8080
    mimetype.assign = (".txt" => "text/plain", ".html" => "text/
   html" )

    server.max-worker = 1
    server.max-connections = 1000
```

This configuration file specifies the server's document root, bind address, port, and maximum number of workers and connections. By defining these parameters, we ensure that the server operates correctly and can handle the incoming requests from the fuzzer.

In particular, we force to have just one worker to avoid problems with fuzzing, because Fallaway is not yet ready to work with multi-process SUT.

3.1.4 Fuzzing Execution and Results

We ran the fuzzer for 24 hours, using the following script to execute it:

```
#!/bin/bash
bin="${1:-mcsm-cy}"
loops="${2:-1000}"

timeout 24h cargo run --release --bin libaflstar-http-$bin -- --
   in-dir /corpus --out-dir /output_lighttpd --target-port 8080
   --loops $loops -t 300 /lighttpd/lighttpd -D -f /lighttpd.conf
```

In particular we have:

- **bin**: the state scheduler strategy.

- **loops**: taken by the __AFL_LOOP, it is the number of iterations that the fuzzer will do.

- **timeout 24h**: the fuzzer will run for 24 hours.

The results of the fuzzing process will be discussed in the chapter 4.

3.2 Comparison with AFLNet and ChatAFL

AFLNet and ChatAFL provide alternative approaches to fuzzing that share certain characteristics with Fallaway, but also have distinct differences in their setup, configuration, and operational strategies. In this section, we outline the setup for both AFLNet and ChatAFL, noting similarities with Fallaway, and highlight the unique features of each tool, including how they handle server responses and their internal mechanisms for optimizing fuzzing performance.

3.2.1 Setting up the fuzzing environment

The setup process for AFLNet and ChatAFL is quite similar to that of Fallaway, given that all three fuzzers share a common Docker-based environment with the necessary dependencies. Both AFLNet and ChatAFL are built and configured using **ProFuzzBench** [4], a benchmark suite specifically designed for evaluating network protocol fuzzers. ProFuzzBench provides a standardized environment and set of targets to ensure a fair comparison among different fuzzers.

By using ProFuzzBench, AFLNet and ChatAFL benefit from a streamlined setup process that automates the installation of dependencies and configuration of the environment, thus reducing setup overhead. This setup also involves additional dependencies, such as specific Python packages, which are necessary for supporting ChatAFL's unique capabilities like leveraging language models internally.

The Docker setup derived from ProFuzzBench provides the same base environment for both AFLNet and ChatAFL, ensuring compatibility and consistency across experiments. By using this common benchmark suite, researchers can directly compare results, further validating the effectiveness and performance differences between the fuzzers. The Dockerfile for AFLNet and ChatAFL is shown from the next page.

```
FROM ubuntu:20.04

# Install common dependencies
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get -y update && \
    apt-get -y install sudo \
    apt-utils \
    build-essential \
    openssl \
    clang \
    graphviz-dev \
    git \
    autoconf \
    libgnutls28-dev \
    libssl-dev \
    llvm \
    python3-pip \
    nano \
    net-tools \
    vim \
    gdb \
    netcat \
    strace \
    libcap-dev \
    libpcre2-dev \
    libpcre2-8-0 \
    libcurl4-openssl-dev \
    libjson-c-dev \
    pkg-config \
    zlib1g-dev \
    automake libtool m4 \
    wget

# Add a new user ubuntu, pass: ubuntu
RUN groupadd ubuntu && \
    useradd -rm -d /home/ubuntu -s /bin/bash -g ubuntu -G sudo -u
     1000 ubuntu -p "$(openssl passwd -1 ubuntu)"

RUN chmod 777 /tmp

RUN pip3 install gcovr==4.2

# Use ubuntu as default username
USER ubuntu
WORKDIR /home/ubuntu

# Import environment variable to pass as parameter to make (e.g.,
     to make parallel builds with -j)
ARG MAKE_OPT
```

```
# Set up fuzzers
COPY --chown=ubuntu:ubuntu aflnet aflnet
RUN cd aflnet && \
    make clean all $MAKE_OPT && \
    cd llvm_mode && make $MAKE_OPT

COPY --chown=ubuntu:ubuntu chatafl chatafl
RUN cd chatafl && \
    make clean all $MAKE_OPT && \
    cd llvm_mode && make $MAKE_OPT

COPY --chown=ubuntu:ubuntu chatafl-cl1 chatafl-cl1
RUN cd chatafl-cl1 && \
    make clean all $MAKE_OPT && \
    cd llvm_mode && make $MAKE_OPT

COPY --chown=ubuntu:ubuntu chatafl-cl2 chatafl-cl2
RUN cd chatafl-cl2 && \
    make clean all $MAKE_OPT && \
    cd llvm_mode && make $MAKE_OPT

# Set up environment variables for AFLNet
ENV WORKDIR="/home/ubuntu/experiments"
ENV AFLNET="/home/ubuntu/aflnet"
ENV PATH="${PATH}:${AFLNET}:/home/ubuntu/.local/bin:${WORKDIR}"
ENV AFL_PATH="${AFLNET}"
ENV AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 \
    AFL_SKIP_CPUFREQ=1 \
    AFL_NO_AFFINITY=1


RUN mkdir $WORKDIR

COPY --chown=ubuntu:ubuntu gcov.patch ${WORKDIR}/gcov.patch

# Set up environment variables for ASAN
env ASAN_OPTIONS=’abort_on_error=1:symbolize=0:detect_leaks=0:
   detect_stack_use_after_return=1:detect_container_overflow=0:
   poison_array_cookie=0:malloc_fill_byte=0:max_malloc_fill_size
   =16777216’

# Download and compile Lighttpd1.4 for fuzzing
RUN cd $WORKDIR && \
    git clone https://git.lighttpd.net/lighttpd/lighttpd1.4.git
   lighttpd1 && \
    cd lighttpd1 && \
    git checkout 9f38b63cae3e2 && \
    ./autogen.sh && \
    CC=afl-clang-fast CXX=afl-clang-fast++ ./configure --enable-
   static --disable-shared && \
    AFL_USE_ASAN=1 make clean all $MAKE_OPT || make all


# Download and compile Lighttpd1.4 for coverage analysis
RUN cd $WORKDIR && \
    git clone https://git.lighttpd.net/lighttpd/lighttpd1.4.git
   lighttpd1-gcov && \
    cd lighttpd1-gcov && \
    git checkout 9f38b63cae3e2 && \
    patch -p1 < ${WORKDIR}/gcov.patch && \
    ./autogen.sh && \
    CFLAGS="-fprofile-arcs -ftest-coverage" CPPFLAGS="-fprofile-
   arcs -ftest-coverage" CXXFLAGS="-fprofile-arcs -ftest-
   coverage" LDFLAGS="-fprofile-arcs -ftest-coverage" ./
   configure --enable-static --disable-shared && \
    make clean all

COPY --chown=ubuntu:ubuntu in-http ${WORKDIR}/in-http
COPY --chown=ubuntu:ubuntu http.dict ${WORKDIR}/http.dict
COPY --chown=ubuntu:ubuntu cov_script.sh ${WORKDIR}/cov_script
COPY --chown=ubuntu:ubuntu run.sh ${WORKDIR}/run
COPY --chown=ubuntu:ubuntu inputs ${WORKDIR}/inputs
COPY --chown=ubuntu:ubuntu lighttpd.conf ${WORKDIR}/lighttpd.conf

RUN cp -rf ${WORKDIR}/inputs/* /tmp/
```

Another important file to consider is the configuration file of the Lighttpd server, which is shown below.

```
    server.document-root = "/tmp"
    server.bind = "127.0.0.1"
    server.port = 8080
    mimetype.assign = (".txt" => "text/plain", ".html" => "text/
   html" )
```

As we can see, in this case we don't have to force the number of workers to 1, because they don't have the same problem as Fallaway, due to their managment of the SUT, explained in the next section.

3.2.2 Mutator and Corpus Management

Similar to Fallaway, AFLNet and ChatAFL use a corpus of test cases to seed the fuzzing process. However, their approach to handling and mutating this corpus differs slightly:

- **AFLNet**: Focuses on network protocol fuzzing by analyzing and mutating protocol-specific fields in input messages. The corpus for

AFLNet includes various protocol messages (e.g., HTTP requests) that are tailored to network targets. AFLNet leverages coverage feedback as well as response error codes from the server to refine its mutations and generate new test cases.

- **ChatAFL**: Enhances the mutation process using a language model (LLM) to generate intelligent mutations. This approach allows it to craft inputs that are more likely to uncover new code paths or trigger complex behaviors. The LLM is used to predict and prioritize inputs based on semantic understanding of the protocol or application under test.

Here are some examples of the corpus used by AFLNet and ChatAFL:

```
GET /hello.txt HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.0.1
Accept: */*
```

Figure 3.3: Seed used by AFLNet and ChatAFL

```
OPTIONS /hello.txt HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.0.1
Accept: */*
```

Figure 3.4: Seed used by AFLNet and ChatAFL

```
DELETE /hello.txt HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.0.1
Accept: */*
```

Figure 3.5: Seed used by AFLNet and ChatAFL

AFLNet and ChatAFL also use a dictionary during fuzzing (Figure 3.6). This dictionary is used to generate meaningful and diverse input cases, ensuring that the fuzzer explores a wide range of scenarios and protocols. By leveraging such a dictionary, these tools enhance their ability to cover different code paths.

```
"GET"
"PUT"
"POST"
"OPTIONS"
"127.0.0.1"
"DELETE"
"CONNECT"
"TRACE"
"HEAD"
"hello.txt"
"User-Agent"
"StarWars3.wav"
```

Figure 3.6: Dictionary used by AFLNet and ChatAFL

3.2.3 Operational Differences: Code Handling and Analysis

One of the main distinctions between Fallaway, AFLNet, and ChatAFL is their strategy for guiding the fuzzing process:

- **AFLNet** and **ChatAFL**: Both tools incorporate error code analysis in their feedback loop. They monitor the response codes (such as HTTP 404, 500, etc.) returned by the server to understand which inputs trigger errors or unexpected states. This allows them to focus on generating inputs that might exploit these observed errors, thereby uncovering potential vulnerabilities.

- **Fallaway**: In contrast, Fallaway exclusively relies on coverage metrics to guide the fuzzing process. It focuses on maximizing the code paths exercised by the generated inputs without directly considering the response codes from the server. This approach enables it to explore new paths more thoroughly, but may overlook specific error states that are of interest for security testing.

3.2.4 Fuzzing Execution and Results

The real difference from Fallaway is in the execution script that we can see here:

```
 #!/bin/bash

 PFBENCH="$PWD/benchmark"
 cd $PFBENCH

 PATH=$PATH:$PFBENCH/scripts/execution:$PFBENCH/scripts/
analysis
 NUM_CONTAINERS=$1
 TIMEOUT=$(( ${2:-1440} * 60))
 SKIPCOUNT="${SKIPCOUNT:-1}"
 TEST_TIMEOUT="${TEST_TIMEOUT:-5000}"

 export TARGET_LIST=$3
 export FUZZER_LIST=$4

 if [[ "x$NUM_CONTAINERS" == "x" ]] || [[ "x$TIMEOUT" == "x"
]] || [[ "x$TARGET_LIST" == "x" ]] || [[ "x$FUZZER_LIST" == "
x" ]]
 then
     echo "Usage: $0 NUM_CONTAINERS TIMEOUT TARGET FUZZER"
     exit 1
 fi

 PFBENCH=$PFBENCH PATH=$PATH NUM_CONTAINERS=$NUM_CONTAINERS
TIMEOUT=$TIMEOUT SKIPCOUNT=$SKIPCOUNT TEST_TIMEOUT=
$TEST_TIMEOUT scripts/execution/profuzzbench_exec_all.sh ${
TARGET_LIST} ${FUZZER_LIST}
```

An example of execution line is like this:

```
./run.sh  <container_number> <fuzzed_time> <subjects> <fuzzers>
```

We can see that the script takes four arguments:

- *CONTAINER_NUMBER*: the number of containers to use for the execution of the fuzzer.
- *FUZZED_TIME*: the time in minutes after which the execution of the fuzzer will be stopped.
- *SUBJECTS*: a list of targets to fuzz.
- *FUZZERS*: a list of fuzzers to use.

We ran the script with the following command (1440 is 24h in minutes):

```
./run.sh 1 1440 lighttpd aflnet,chatafl
```

Results from both AFLNet and ChatAFL will be discussed in detail in Chapter 4.

4 Results

- Fallaway:

```
complete_coverage: 6% (883/15168)
total_executions: 79987763
```

- AFLNet:

```
complete_coverage: 5% (830/15168)
total_executions: 209776
```

- ChatAFL:

```
complete_coverage: 6% (850/15168)
total_executions: 241242
```

After 24h of fuzzing, we can look at the results and analyze them.
Fallaway has the highest coverage of 6% with 883 edge cases covered.
It has the highest number of total executions with 79,987,763, which is 381 times more than **AFLNet** and 332 times more than **ChatAFL**.
AFLNet has the lowest coverage of 5% with 830 edge cases covered.
It has the lowest number of total executions with 209,776.
ChatAFL has a coverage of 6% with 850 edge cases covered.
It has a total of 241,242 executions.
The coverage of the three fuzzers over 24h is shown in Figure 4.1.

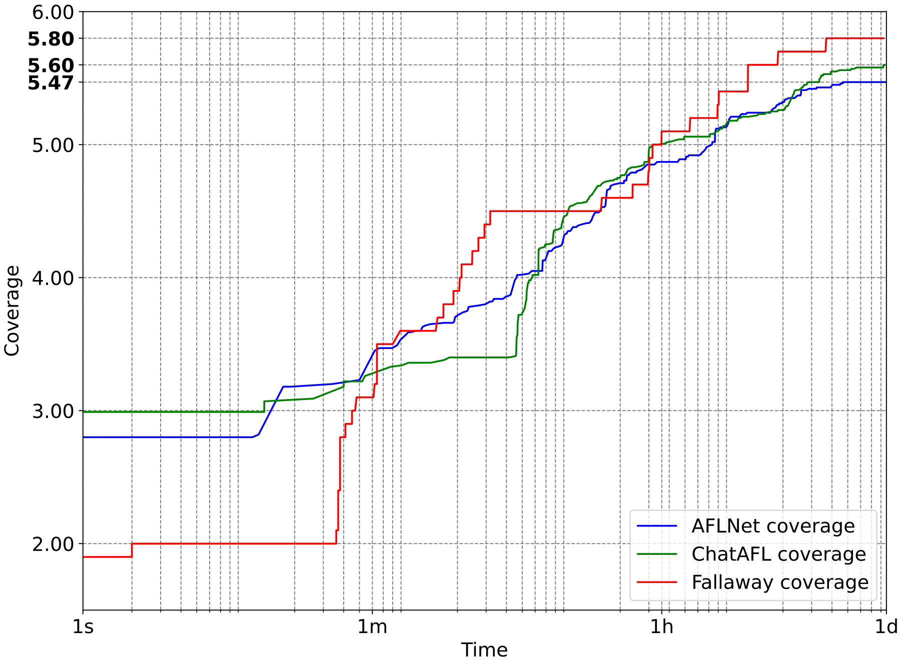

Figure 4.1: Coverage of the three fuzzers in 24h

In another experiment, we ran the fuzzers for 10 hours, modifying the configuration of Fallaway to be in a loop of 250.
The coverage of the three fuzzers over 10h is shown in Figure 4.2.

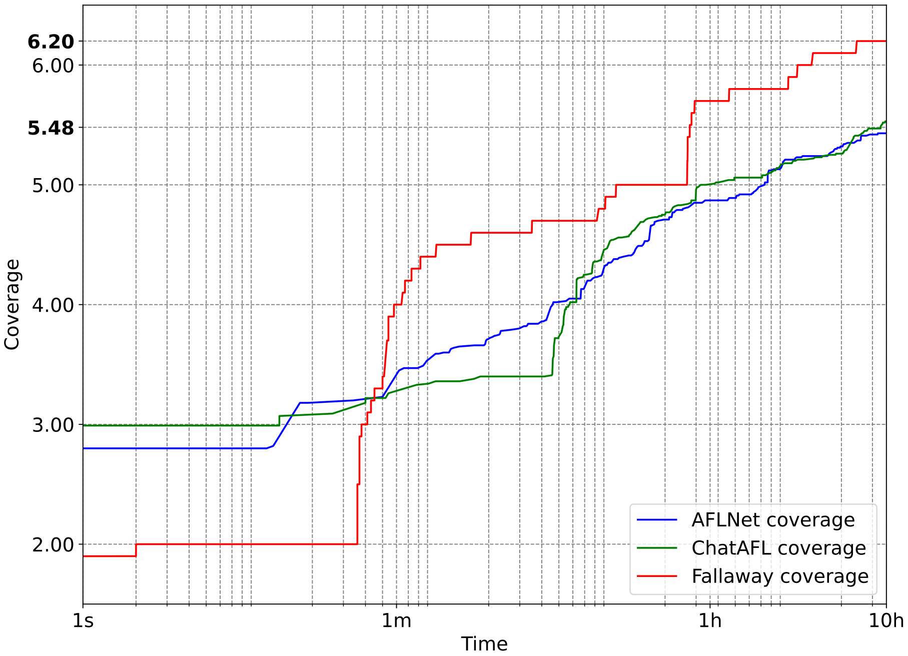

Figure 4.2: Coverage of the three fuzzers in 10 hours

In this configuration, Fallaway has reached a complete_coverage of 6% (942/15168).
This is 59 more edge cases than the previous configuration, which is a 6.7% increase.
This could be due to the fact that the fuzzer is running in a loop of 250, which allows it to run more test cases in a shorter amount of time.
This is not always a good thing, it depends on the SUT, corpus, configurations and more because in general: higher loop values can lead to more coverage, but also to more redundant test cases and to stuck on states that makes low progress; lower loop values can lead to less coverage, but also to less redundant test cases and to faster progress.

5 Conclusion

Conclusions

6 Future Works

6.1 Fuzzing with Reinforcement Learning

Although Fallaway has shown promising results in our experiments, there is still room for improvement. One of the most viable approaches to achieving an optimal configuration involves the use of Reinforcement Learning (RL).

Reinforcement Learning is a subfield of machine learning in which an agent learns to make decisions through interaction with an environment. During this interaction, the agent performs a series of actions and receives rewards or penalties based on those actions. The agent uses this feedback to learn an optimal policy — a strategy for selecting actions that maximizes the cumulative reward over time. A key concept in RL is the trade-off between exploration (trying new actions to discover their effects) and exploitation (choosing known actions that yield high rewards).

By applying RL to fuzzing, the system can automatically adapt parameters such as mutation rates, seed selection strategies, and other relevant configurations to maximize code coverage. In this context, the RL agent learns to identify and prioritize inputs that are likely to trigger new edge cases or explore untested paths in the code. Each time the fuzzer discovers a new edge case or covers a previously unexplored branch, it receives a reward. Over time, the agent refines its strategy to generate inputs that are more effective at increasing coverage, thereby enabling a more comprehensive exploration of the software under test.

Integrating RL into the configuration process of fuzzers allows for continuous optimization of the fuzzer's behavior at runtime. This dynamic adaptation can significantly enhance the fuzzer's ability to find vulnerabilities and other critical software defects.

6.2 Fuzzing with multi-process SUTs

Another one interesting future avenue of work is to extend Fallaway for fuzzing multi-process SUTs.
There are already some works on this field, let me cite some: *Evaluating the Fork-Awareness of Coverage-Guided Fuzzers* [5], *Forkfuzz: Leveraging the Fork-Awareness in Coverage-Guided Fuzzing* [6].
In many modern software systems, most of which are designed to provide high performance and scalability, different tasks or their stages of execution in most cases are performed by multiple processes in cooperation. Web servers (like even Lighttpd) are an example where a main process would listen constantly for incoming network connections while spawning child processes with every request handled separately. That architectural pattern allows improvement in both responsiveness and fault isolation; it also presents new challenges when performing fuzzing.
Fuzzing multi-process systems typically relies on crafting inputs that effectively exercise the communication and interactions between the different processes. This becomes more complex than pure single-process fuzzing, as the fuzzer has to orchestrate the execution of multiple processes together: it needs to make sure that inputs are correctly synchronized and passed between the various processes running in parallel or at a different stage of execution.
To this end, the fuzzer would need to understand not only the data that flows between the processes involved but also the timing and state dependencies inherent in such a setup. In other words, it needs to generate inputs that trigger specific sequences of interactions between the main process and its spawned child processes or between processes that talk to each other through shared memory, sockets, or any of the other IPC mechanisms available.
For Fallaway to handle such cases, further extension would be needed for increased multi-process tracking and management of the execution flow together with monitoring and manipulating effectively the IPC channels. It might include developing hooking techniques into the paths of communications, capturing and mutating the messages which processes exchange or introducing small delays under control in order to explore different interleavings of executions. Overcoming this would allow Fallaway to fuzz modern software systems more comprehensively, which could be exposing bugs that only appear under multiprocessor environments.

References

[1] 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.

[2] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[3] Timme Bethe. Fallaway: High throughput stateful fuzzing by making afl* state-aware. 2024.

[4] 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.

[5] Marcello Maugeri, Cristian Daniele, Giampaolo Bella, and Erik Poll. Evaluating the fork-awareness of coverage-guided fuzzers. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy - ICISSP*, 2023.

[6] Marcello Maugeri, Cristian Daniele, and Giampaolo Bella. Forkfuzz: Leveraging the fork-awareness in coverage-guided fuzzing. In *Computer Security. ESORICS 2023 International Workshops: CPS4CIP, ADIoT, SecAssure, WASP, TAURIN, PriST-AI, and SECAI, The Hague, The Netherlands, September 25–29, 2023, Revised Selected Papers, Part II*, 2023.