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*, executions and crash detection.
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. Taking some examples of these technologies, there is **SAGE** [1], developed by Microsoft, which introduced whitebox fuzzing by using dynamic symbolic execution to explore different execution paths in software. This method significantly contributed to identifying vulnerabilities in Windows by systematically generating inputs that maximize code coverage.
Another example is **ClusterFuzz** [2], developed by Google, which is a large-scale fuzzing infrastructure that automates the testing of software like the Chrome browser. It has been instrumental in identifying thousands of security vulnerabilities by continuously running a diverse set of fuzzers, including those based on coverage-guided techniques.
Furthermore, there is American Fuzzy Lop (**AFL** [3]), which introduced a new approach with coverage-guided fuzzing. This method uses feedback from program execution to guide the mutation of inputs, focusing on maximizing code coverage rather than generating inputs randomly.
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 [4]. 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 project is **Lighttpd** [5], an *open-source web server* recognized for its effectiveness and ability to scale, to manage a substantial number of concurrent connections. Assessing Lighttpd offers a chance to scrutinize stateful fuzzing methodologies.
This project focuses on the benchmarking of stateful fuzzers to ascertain their efficiency in *coverage* in Lighttpd. The reviewed 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, this project will report the various strengths and weaknesses of each, which gives necessary suggestions for improving stateful fuzzing techniques and enhancing the security of modern software systems.
The project is structured as follows: Chapter 2, provides an overview of the background about fuzzing, fuzzers used in this project and the target. Chapter 3, describes the setup of the environment and the configuration of the fuzzers. Chapter 4, presents the results obtained from the experiments.
Finally, Chapter 5, summarizes the findings and provides suggestions for future work.

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 [6].
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.

2.1.1 Types of Fuzzing Techniques

Fuzzing methodologies vary and there exist a lot for different applications and purposes [7]:

- **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 [8].
- **Grey-box Fuzzing**: It is 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. Grey-box fuzzing has become a widespread technique with the advent of **AFL** tool (*American Fuzzy Lop* [3]), which has proved to be highly effective using coverage-guided fuzzing, as explained in Section 2.1.3

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 weaker compared with other generations for applications requiring highly structured inputs [9].
- **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 [9].

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** [3]) 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 (see 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**, where each edge is mapped to a bit in the coverage map.

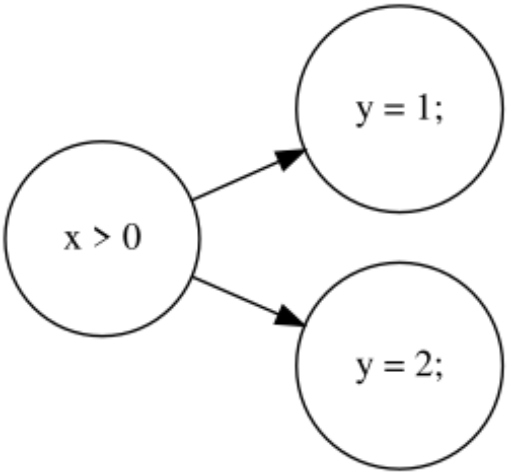

Figure 2.1: Example of edge between basic blocks

When an edge is executed, the corresponding bit in the coverage map is incremented by *+1* (to mark as "*hitted*"). 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.
- **Interesting inputs**: Inputs that are generated by the fuzzer during the fuzzing process and achieve new coverage (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.

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 considering an FTP server like **LightFTP** [10], 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.

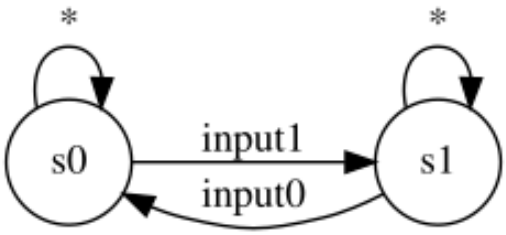

Figure 2.2: A simple state model illustrating state transitions in a stateful application

Considering this, from state *s0* there is a transition to state *s1* only if the input is a certain input "*input1*". On the other hand, from state *s1* there is also a transition to state *s0* only if the input is a certain input "*input0*". If any other input is given, the state remains the same.

In this case, the fuzzer should be able to generate a sequence of inputs (*input0, input1 and other inputs*) that first brings the application from state s0 to state s1 and then tests the behavior of the application in that state (and vice versa).

2.2.2 Key Techniques in Stateful Fuzzing

Stateful fuzzing involves several advanced techniques that distinguish it from traditional fuzzing approaches [7]:

- **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.
- **Feedback Mechanisms**: By tracking the state of the application and taking feedbacks, 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 [11, 12].

2.2.3 Challenges in Stateful Fuzzing

Stateful fuzzing has also some challenges to face [4]:

- **Complex State Transitions:** Stateful systems often have intricate state transitions. To perform fuzz testing effectively, the system must be guided through various states. Using the same input specification for different states may lead to many packets being discarded if they do not conform to the expected format, thereby reducing the effectiveness of the test cases [13].
- **Resource Consumption:** Protocol messages require significant time and memory resources for reception, response, and transmission between the client and server. As the depth of state transitions increases,

the number of auxiliary packets transmitted and memory resources consumed grows exponentially, resulting in a lower number of test packets processed per unit of time. This impact on resource usage can significantly reduce testing efficiency [13].

- **Path Explosion**: The number of possible paths through the application increases exponentially with the number of states and transitions. This can lead to a combinatorial explosion of possible paths, making it difficult to explore all possible states and transitions effectively [14].

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 Git [15, 16].

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 maintains transient states during the processing of requests as shown if Figure 2.3.

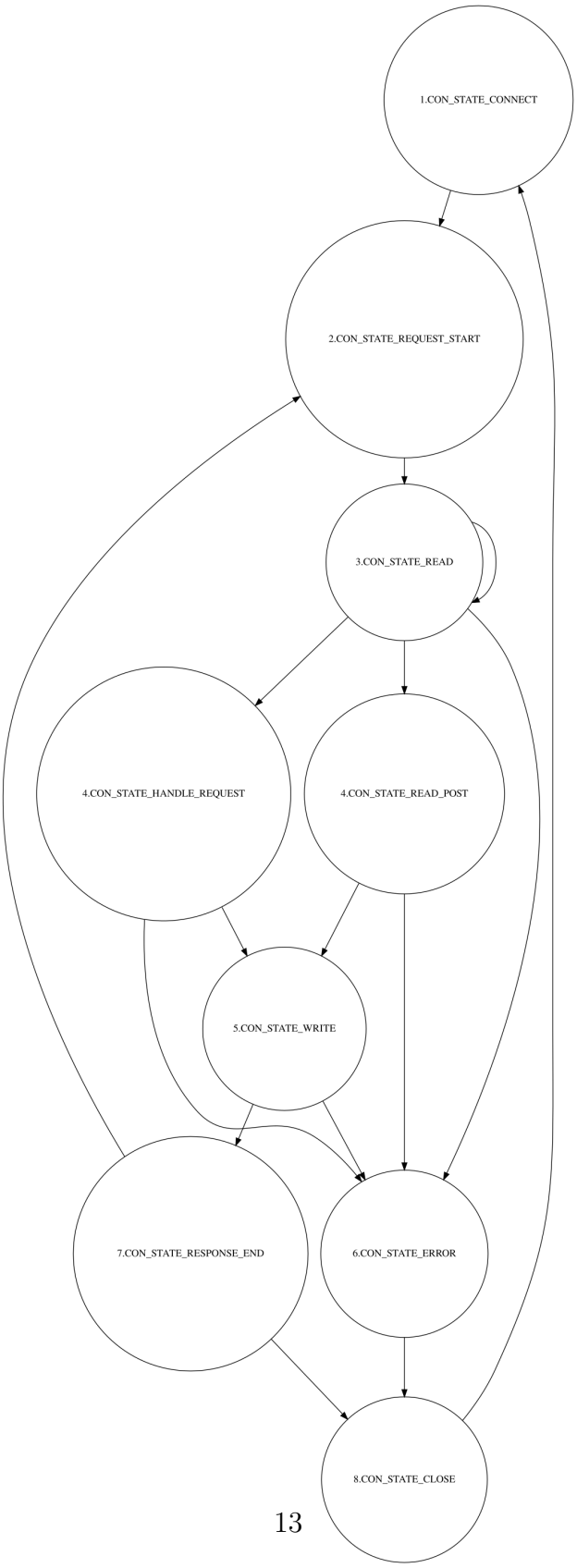

Figure 2.3: State model of Lighttpd

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

1. **CON_STATE_CONNECT**: The initial default state before a connection is established.
2. **CON_STATE_REQUEST_START**: The state right after a connection is established and wait for request.
3. **CON_STATE_READ**: The state when the server is reading the request (here the server can be in a loop to read the request).
4. **CON_STATE_HANDLE_REQUEST**: The state when the server is processing the request, if body length is null.
5. **CON_STATE_READ_POST**: The state when the server is processing the request, if body length is more than 0.
6. **CON_STATE_WRITE**: The state when the server is writing the response.
7. **CON_STATE_ERROR**: The state when an unhandled error occurs.
8. **CON_STATE_RESPONSE_END**: The state after the request has been fully received.
9. **CON_STATE_CLOSE**: The state when the connection is closed.

The *CON_STATE_CONNECT* is reached just 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 the request is sent line by line, it will loop into it).
The other states are transitory because, ultimately, the connection will go back to *CON_STATE_REQUEST_START* or *CON_STATE_CONNECT*.
For this project, it has been chosen to model the state of the server based on the existence or non-existence of a resources. A resource can be a file, a directory, or any other entity that can be accessed via HTTP. By considering two types of state—a state *s0*, where the resource does not exist and a state *s1* where the resource exists—it can effectively explore different states of the server.

2.4 Benchmark Selection: AFLNet, ChatAFL and Fallaway

For the scope of this project, three stateful fuzzers—AFLNet, ChatAFL and Fallaway—will be benchmarked over Lighttpd.
AFLNet was chosen as the initial CGF (coverage-guided fuzzer) because it is the first in its category [17], while ChatAFL and Fallaway, being the most recent stateful fuzzer tools published this year, have achieved significant advancements over AFLNet in some case studies.

2.4.1 AFLNet

AFLNet [17] 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 seeds 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, which overall algorithm can be summarized as follows [18]:

1. Load user-supplied initial test cases into the queue.
2. Take next input file from the queue.
3. Attempt to trim the test case to the smallest size that does not alter the measured behavior of the program.
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies.
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

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)* [19] 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 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 [20] , 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 [21] is a stateful fuzzer designed to address several key challenges faced by traditional fuzzers when handling stateful SUTs. Unlike stateless fuzzers such as AFL, which send single test cases and expect the SUT to terminate, Fallaway manages multiple states by incorporating a *dual-loop* structure: an outer loop that selects the SUT 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 scheduling.
State scheduling is based on different strategies like:

- **Coverage-Yield (CY)** strategy, which uses a round-robin approach to select the next state.
- **Outgoing Edges (OE)** strategy, which selects states prioritizing the outgoing edges of the state model.

Corpus and queue are often 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.
About test case and coverage map, they can be state aware or not. Being state aware means that they are specific to a state, otherwise they are shared across all states.
Fallaway uses two different strategies to manage the corpus:

- **Multiple Corpus Single Map (MCSM)**: This strategy uses a single coverage map, shared accross the states, to track the coverage of the program. It also uses multiple queues, one for each state, to store the inputs.
- **Multiple Corpus Multiple Map (MCMM)**: This strategy uses multiple coverage maps, one for each state, to track the coverage of the program and multiple queues, one for each state, to store the inputs.

When the state scheduler choose a state, that specific queue will contains "*interesting*" inputs, that have allowed the fuzzer to achieve more coverage. For each state, a unique prefix is maintained along with a separate corpus of test cases, allowing focused exploration of the SUT'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** [22] framework, which is a modular library for developing fuzzers. To make LibAFL suitable for stateful SUTs, Fallaway extends its functionality in two key ways:

1. It uses AFL's **persistent mode**, designed to keep a SUT 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 SUT 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 SUT 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, it is necessary to 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. It is also important 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, it has been defined the state of the server based on the existence 3.1 or non-existence 3.2 of a resource. Specifically

considering 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** folder consists of: a set of folders, one for each state, each containing a set of files, the *prefixes*, that are a sequence of messages to reach that state.
In this case, the corpus is composed by two folders: one for the existent resource and one for the non-existent resource. Each folder contains a single file with a request to reach the state. Here we have two requests for the two states:

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

Hello, World!
```

Figure 3.1: 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: Non-existent resource request

The **mutator**, responsible for generating variations of the requests, is relatively straightforward. Its primary function is to modify the existing requests by appending the sequence of characters "$\backslash r \backslash n \backslash r \backslash n$" to the end of each request. This modification is essential as it ensures that the requests are well-formed and adhere to HTTP protocol standards.
Before this, the mutator adds a set of tokens (taken from files in the corpus folder, but outside of the state folder) and places them in random positions within the request.
By ensuring the requests are properly formatted, the mutator enables the

server to parse and process them correctly, which is vital for accurate fuzz testing.
An example of the corpus folder is as follows:

```
corpus
|-- existent_resource
|   |-- 0_put
|   |-- metadata
|-- non_existent_resource
|   |-- 0_delete
|   |-- metadata
|-- GET
|-- DELETE
|-- PUT
|-- OPTIONS
|-- POST
```

Figure 3.3: Corpus folder structure

Summing up: *existent_resource* folder is a state, *0_put* is a prefix to reach that state and, in this case is the full request to reach it. The same for *non_existent_resource* and *0_delete*. The *metadata* files contains the number of outgoing edges for that state. In this case, the number of outgoing edges is 2 for both states, because we can have:

- *existent_resource* state: when sending the put request, the server can return a 404 error, if the resource exists, or can return a 200 OK.
- *non_existent_resource* state: when sending the delete request, the server can return a 404 error, if the resource does not exist, or can return a 200 OK.

Finally the other files rapresent the tokens that the mutator will use to generate new requests.

3.1.3 Setting Up the Fuzzing Environment

To run the fuzzer, it is important to build a Docker container that includes all the necessary dependencies and the modified Lighttpd server. The Dockerfile below is based on an image that already contains Fallaway and shows the steps to set up this environment.

```
FROM fallaway

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 /Fallaway/run.sh
# Make it executable
RUN chmod +x /Fallaway/run.sh

WORKDIR /Fallaway
```

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 the server will operates correctly and can handle the incoming requests from the fuzzer.
In particular, it is forced to have just one worker to avoid problems with fuzzing, because Fallaway is not designed to work with multi-process SUT.

3.1.4 Fuzzing Execution and Results

To run the fuzzer for 24 hours, it is necessary to run the following script:

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

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

In particular there are:

- **timeout 24h**: a timeout of 24h for the next command.
- **cargo run**: the command to run the fuzzer.
- –**release**: the flag to run the fuzzer in release mode.
- –**bin fallaway-http-$bin**: the state scheduler strategy (by default is mcsm-cy).
- –: the flag to separate the fuzzer arguments from the binary arguments.
- –**in-dir /corpus**: the input directory for the fuzzer.
- –**out-dir /output_lighttpd**: the output directory for the fuzzer, where the results will be stored.
- –**target-port 8080**: the port of the server.
- –**loops $loops**: the number execution the fuzzer will do before changing state (the $\_\_AFL\_LOOP$ is bigger volountarly, so that we prioritize this argument).
- **-t 300**: the timeout for each test case, in milliseconds, which will trigger if the fuzzer does not reach the end of the $\_\_AFL\_LOOP$ in time.
- **/lighttpd/lighttpd -D -f /lighttpd.conf**: the command to run the server, in detached mode, with the configuration file.

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. The purpose of this section is to 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** [23], 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. Another important thing 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 seen above, in this case there is no need to force the number of workers to 1, because they do not 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 3.4: 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 3.5: 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 3.6: Seed used by AFLNet and ChatAFL

AFLNet and ChatAFL also use a dictionary during fuzzing, shown in Figure

3.7. 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 3.7: 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:

```
#!/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}
```

This is the script defined by ChatAFL repository [24], that internally uses ProFuzzBench's scripts to run the fuzzers [25].
An example of execution line is like this:

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

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.

The command to fuzz Lighttpd for 24h using both AFLNet and ChatAFL is:

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

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

4 Results

The Docker containers, running on an *HP ProLiant DL380p Gen8* server with *32 cores* at *2.70 GHz* and *175.97 GB of RAM*, were configured to execute AFLNet, ChatAFL, and various Fallaway configurations for 24 hours, sharing the same resources.
After 24h of fuzzing, we can look at the results and analyze them, but before this, another step has been done to obtain the coverage of AFLNet and ChatAFL. By default, the output of AFLNet and ChatAFL is a *gcov* file, giving line and branch coverage.
Using the **replayer** (a component designed to replay test cases or inputs, generating a coverage report), we have replicated the queue of AFLNet and ChatAFL and replayed it to calculate the complete coverage.
In Table 4.1 we can see the comparison of the coverage and total executions for Fallaway, AFLNet, and ChatAFL.
Fallaway has been run with different loop configuration (this is explicited by the number after *"Fallaway"* in the table). The strategy used is the *mcsm-oe* (*Multiple Corpus Single Map - Outgoing Edges*), because it has shown to be better than the other strategies [21].
There are also other two columns, *"> AFLNet"* and *"> ChatAFL"*, that show the time in which the fuzzer overcame the coverage of that column's fuzzer.
In bold we are highlighting:

- The highest and lowest coverage.
- The highest and lowest time to overcome AFLNet.
- The highest and lowest time to overcome ChatAFL.
- The highest and lowest number of total executions.

4.1 Fuzzer Analysis

Fuzzer	Complete Coverage	> AFLNet	> ChatAFL	Total Executions *
Fallaway 2000	**5.96% (905/15168)**	2h 56m 24s	**6h 25m 48s**	**120,726,564**
Fallaway 1000	5.82% (883/15168)	3h 22m 48s	3h 35m 24s	79,987,763
Fallaway 500	5.88% (893/15168)	**0h 45m 23s**	**1h 48m 0s**	81,231,325
Fallaway 250	5.87% (891/15168)	0h 51m 16s	2h 9m 0s	96,929,952
Fallaway 100	5.91% (896/15168)	4h 17m 24s	4h 43m 48s	59,916,978
Fallaway 10	**4.89% (741/15168)**	Nan	Nan	18,892,551
AFLNet	5.47% (830/15168)	Nan	Nan	**209,776**
ChatAFL	5.60% (850/15168)	**9h 16m 48s**	Nan	241,242

Table 4.1: Comparison of Coverage and Total Executions for Fallaway, AFLNet, and ChatAFL (Fallaway has been run with different loop values).
***** Take a look at Section 4.2.1 for a deep understanding of this value.

4.2 Coverage Analysis Over Time and Configurations

In the next figures, we can see the coverage of the three fuzzers over time.

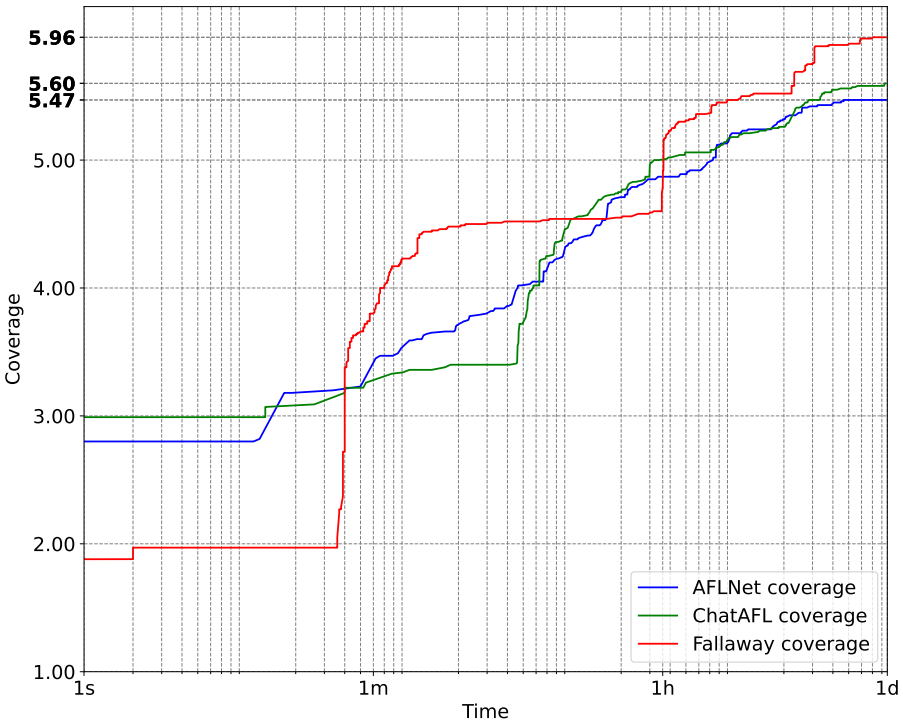

Figure 4.1: Coverage of the three fuzzers in 24h with Fallaway in a loop of 2000

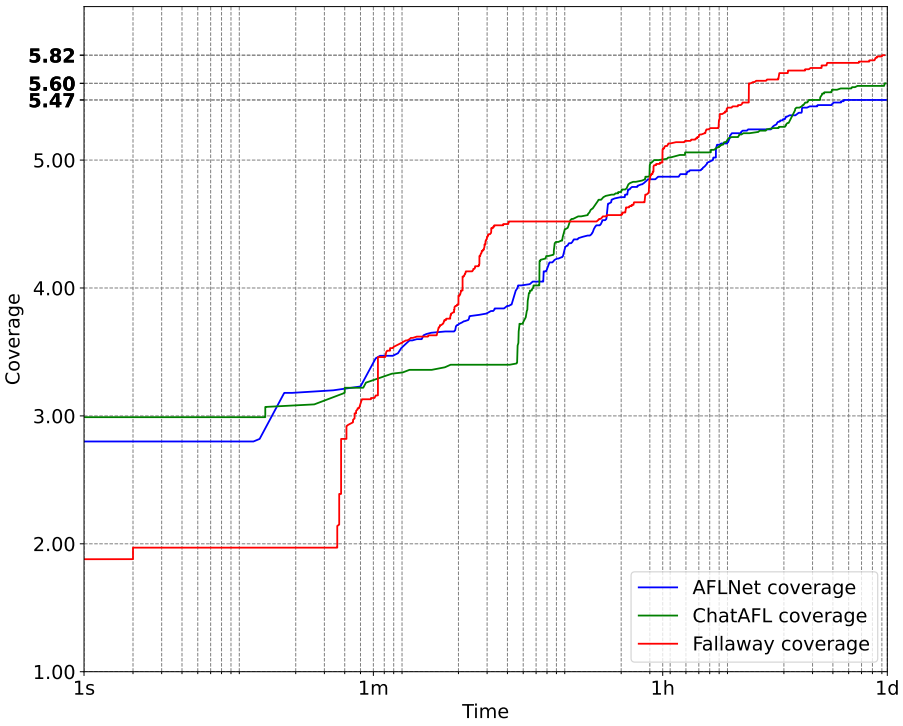

Figure 4.2: Coverage of the three fuzzers in 24h with Fallaway in a loop of 1000

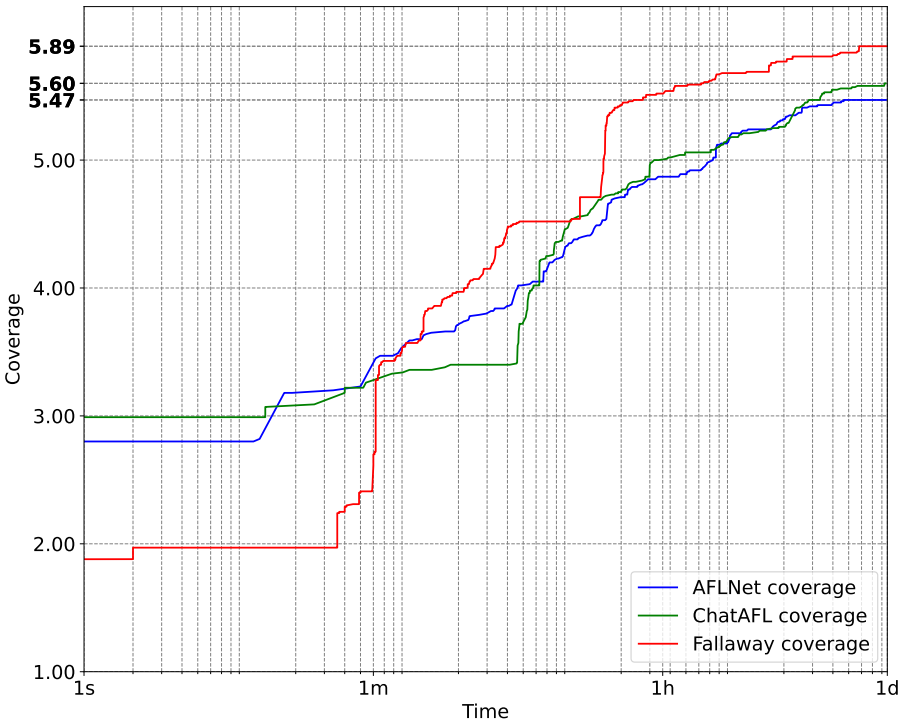

Figure 4.3: Coverage of the three fuzzers in 24h with Fallaway in a loop of 500

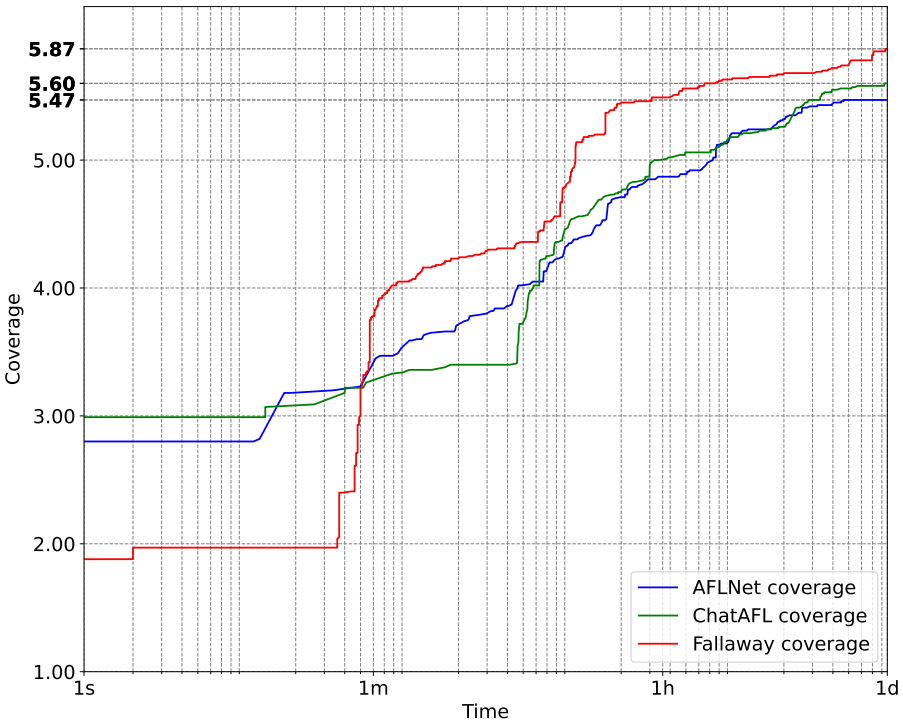

Figure 4.4: Coverage of the three fuzzers in 24h with Fallaway in a loop of 250

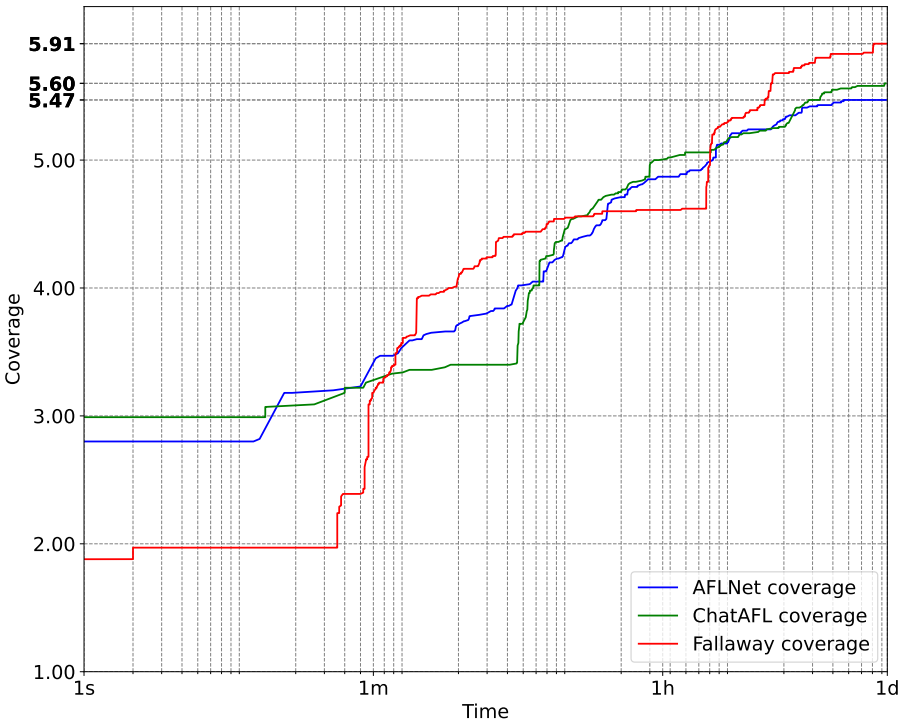

Figure 4.5: Coverage of the three fuzzers in 24h with Fallaway in a loop of 100

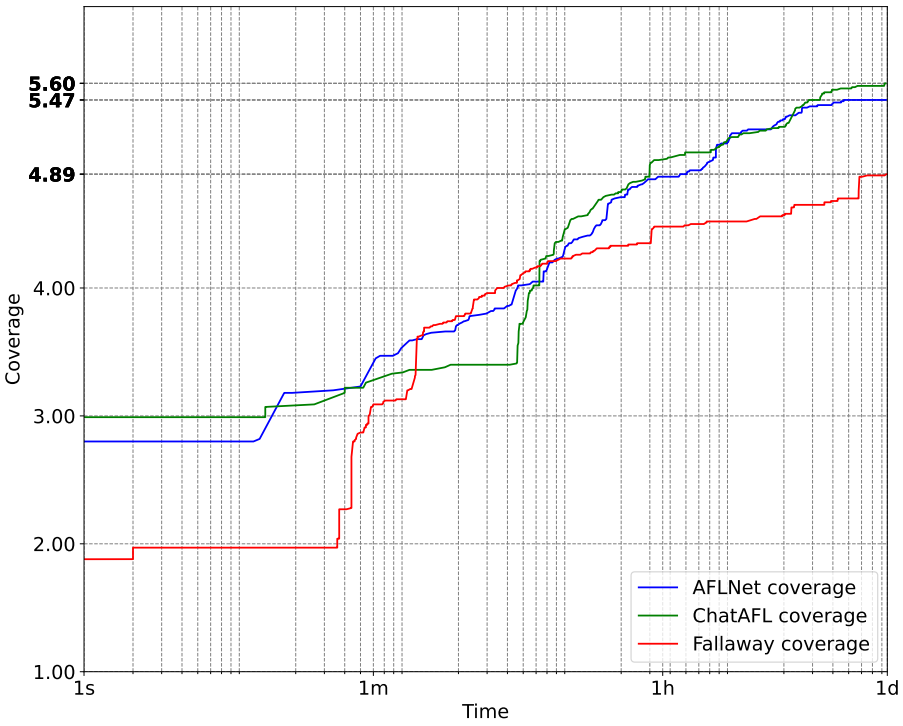

Figure 4.6: Coverage of the three fuzzers in 24h with Fallaway in a loop of 10

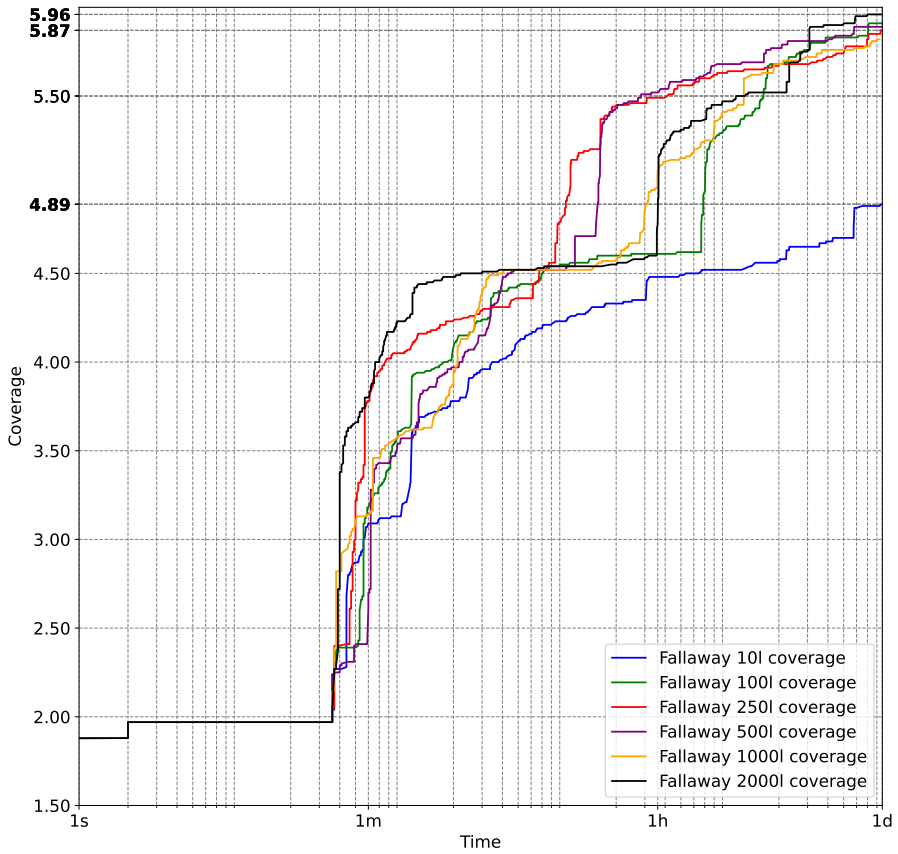

Figure 4.7: Coverage of Fallaway's differents configurations in 24h

As shown in Figure 4.1, Fallaway with a loop count of 2000 achieves the highest coverage. This is likely because the fuzzer can execute more test cases before transitioning between states. The other configurations of Fallaway, except for the one with 10 loops, share a lower coverage than the 2000 loops' case, having an average coverage of 5.87%.
Although the loop counts differ, the coverage does not change significantly. This may be due to the implemented state model, which only considers two states: the existence or non-existence of a resource. As a result, even if the number of loops — and thus the number of possible test cases before changing states — varies, the coverage remains relatively stable. This limitation

could be addressed by defining a more refined state model that includes additional states.
Figure 4.6 shows the lowest coverage at 4.89%. In this scenario, the low loop count suggests a minimal number of possible test cases before switching states, which reduces the likelihood of extensive exploration.
In general, higher loop values can lead to greater coverage by exploring more states, but they may also cause deadlock states where progress is limited. Conversely, lower loop values can help overcome deadlock states more quickly, but they might not explore certain states in depth.
Fallaway does not always exhibit linear growth in coverage; instead, it shows a more random behavior, even though its approach yields the best results observed. In contrast, AFLNet and ChatAFL demonstrate more linear growth in coverage, which indicates greater efficiency in code exploration. They start with higher initial coverage than Fallaway, suggesting they are better at targeting diverse code paths.
The linearity is shown using the *Pearson correlation coefficient* in Table 4.2. Pearson correlation coefficient measures the strength and direction of a linear relationship between two variables (in this case, time and coverage). The closer the coefficient is to 1, the stronger the linear relationship is. In this case, the coefficient for AFLNet and ChatAFL is 0.67, indicating a good linear relationship between time and coverage.
Fallaway's coefficient varies, depending on the loop count, with the highest value of 0.71 for 100 loops and the lowest value of 0.47 for 500 loops. This suggests that Fallaway with 100 loops has the most linear relationship between time and coverage, instead of the 500 loops configuration, which has the lowest linear relationship.
Regarding coverage, AFLNet achieves a coverage of 5.47% with significantly fewer executions compared to the closest coverage for Fallaway's configurations (1000 loop, see Table 4.1).
ChatAFL reaches 5.60% coverage with a number of executions similar to AFLNet. As ChatAFL is based on AFLNet, it effectively surpasses the coverage plateau observed with AFLNet. In particular, as shown in Figure 4.6, which minimizes the interference from Fallaway, ChatAFL surpasses AFLNet's coverage plateau most of the time. Additionally, within the 1-day timeframe, it exhibits a slight increase in coverage, indicating that it was able to exceed its own previous plateau.

Fuzzer	Pearson Correlation Coefficient
Fallaway 2000	0.67
Fallaway 1000	0.64
Fallaway 500	0.47
Fallaway 250	0.53
Fallaway 100	0.71
Fallaway 10	0.54
AFLNet	0.67
ChatAFL	0.67

Table 4.2: Pearson Correlation Coefficient for the three fuzzers

4.2.1 Fallaway

Strengths:

- *Higher Execution Count*: Fallaway has a significantly higher number of total executions compared to AFLNet and ChatAFL, indicating its capability to generate and execute a large number of test cases. This increases the likelihood of discovering bugs or vulnerabilities through extensive input space exploration. An important distinction is that Fallaway treats each execution as a single input, whereas AFLNet and ChatAFL treat executions as full *traces*—sequences of inputs, typically 5 or 6, sent before restarting the SUT's process. Therefore, the adjusted number of executions for a fair comparison with Fallaway is 1,258,656 for AFLNet and 1,447,452 for ChatAFL. Even with this adjustment, Fallaway still has a higher number of executions.
- *Higher Code Coverage*: Achieves the highest code coverage among the three fuzzers (5.96%), suggesting its strategy, even if may lacks of sophistication in targeting specific code paths relies on brute force to uncover edge cases, is effective at exploring various parts of the code.

Weaknesses:

- *Efficiency Concerns*: The large number of total executions (nearly 120 million in the best case) suggests that Fallaway may be less efficient, requiring more attempts to cover similar amounts of code compared to AFLNet and ChatAFL.

4.2.2 AFLNet

Strengths:

- *Efficient Execution*: With the lowest number of total executions (around 210,000), AFLNet is highly efficient in achieving its results. This suggests AFLNet is effective in targeting specific code parts with minimal test cases, leveraging coverage-guided strategies.

Weaknesses:

- *Lower Code Coverage*: AFLNet achieves slightly lower coverage (5.47%) compared to Fallaway and ChatAFL, indicating it might not explore as many diverse code paths outside of network protocol contexts.

4.2.3 ChatAFL

Strengths:

- *Balanced Approach*: ChatAFL exhibits a balance between execution count and coverage. With a small number of executions (around 241,000) and a good code coverage (5.60%), it balances efficiency and effectiveness.

Weaknesses:

- *Potential Limitations in General Fuzzing Tasks*: While optimized for certain scenarios, its overall effectiveness may vary depending on the specific use case and target application. Even though it performs many more state changes [20], it does not achieve a significant increase in coverage in this case study.

5 Conclusion

This project explored the effectiveness of three stateful fuzzing tools—Fallaway, AFLNet, and ChatAFL—when applied to the Lighttpd web server. The study aimed to evaluate each tool, focusing on aspects like coverage, efficiency, and adaptability.

The importance of stateful fuzzing has been discussed, particularly for applications where internal states and state transitions significantly impact behavior, such as web servers and network-based applications. Stateful fuzzing techniques were shown to be crucial in effectively testing these SUTs, as they consider the influence of previous interactions on the application's current state.

The research also involved comparing the setup processes for each fuzzer. Each tool required a different configuration and environment setup to achieve optimal performance. For instance, Fallaway's modifications to Lighttpd required a persistent mode to maintain server states across multiple requests. ChatAFL leverages large language models (LLMs) for generating inputs, while AFLNet utilizes a network-aware approach to enhance its fuzzing capabilities.

Additionally, various graphs were presented to illustrate the comparative results of the fuzzers. These showed differences in execution counts, code coverage achieved over time, and other performance metrics, providing a clearer understanding of how each tool behaves in different scenarios.

Fallaway was observed to extensively explore possible program behaviors, achieving broad coverage by conducting numerous test executions. Its method is advantageous in scenarios where a comprehensive examination of all potential states is crucial.

On the other hand, **AFLNet** leveraged its specialization in network

protocols to achieve meaningful results with fewer test cases. It effectively targeted specific parts of the code, making it suitable for applications that require testing of network-related functionalities. However, its narrower focus might limit its applicability to broader testing needs.

ChatAFL employs an innovative approach by utilizing advanced techniques to enhance input generation. By leveraging large language models (LLMs) to generate inputs and effectively overcoming coverage plateaus, it offers a strong solution for fuzzing. However, it remains somewhat less effective compared to Fallaway.

The findings show that Fallaway has achieved more results, despite its relatively simpler approach. The persistent mode significantly contributes to its effectiveness and the knowledge of the state model in Lighttpd, even if it is somewhat limited due to the absence of real proper well-defined states, still enables Fallaway to produce better results.

5.1 Future Works

Future work could explore combining the efficiency of persistent mode with approaches based on large language models (LLMs).
Another potential area of investigation is fuzzing Lighttpd in a multiprocess setting, without limiting the number of worker processes to one. This would require developing or employing fork-aware fuzzing techniques [26, 27], which is left as future research.

References

[1] P. Godefroid, M.Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. 2012.

[2] Zhen Yu Ding and Claire Goues. An empirical study of oss-fuzz bugs. 2021.

[3] American Fuzzy Lop (AFL) Website. `https://lcamtuf.coredump.cx/afl/`.

[4] Cristian Daniele. Is stateful fuzzing really challenging?, 2024.

[5] Lighttpd GitHub Repository. `https://github.com/lighttpd`.

[6] Zhenhua Yu, Zhengqi Liu, Xuya Cong, Xiaobo Li, and Li Yin. Fuzzing: Progress, challenges, and perspectives. 2024.

[7] Cristian Daniele, Seyed Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. 2024.

[8] Patrice Godefroid, Adam Kiezun, and Michael Levin. Grammar-based whitebox fuzzing. 2008.

[9] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. 2018.

[10] LightFTP GitHub Repository. `https://github.com/hfiref0x/LightFTP`.

[11] Bernhard Aichernig, Edi Muskardin, and Andrea Pferscher. Learning-based fuzzing of iot message brokers. 2021.

[12] Hao Cheng, Dongcheng Li, Man Zhao, Hui Li, and W. Wong. A comprehensive review of learning-based fuzz testing techniques. 2024.

[13] Quanyu Tao. Gonet: Gradient oriented fuzzing for stateful network protocol, 2023.

[14] Path Explosion. https://en.wikipedia.org/wiki/Path_explosion.

[15] Lighttpd Website. https://www.lighttpd.net/2007/4/4/powered-by-lighttpd/.

[16] Lighttpd Wikipedia Page. https://en.wikipedia.org/wiki/Lighttpd.

[17] 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.

[18] AFL Approach. https://afl-1.readthedocs.io/en/latest/motivation.html#the-afl-approach.

[19] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2024.

[20] 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.

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

[22] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. 2022.

[23] 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.

[24] Chatafl GitHub Repository. https://github.com/ChatAFLndss/ChatAFL/tree/master.

[25] Chatafl GitHub ProFuzzBench. https://github.com/ChatAFLndss/ChatAFL/tree/master/benchmark.

[26] 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.

[27] 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.