Design, Implementation and Evaluation of State dependent Fuzzing

An approach of stateful fuzzing to improve code coverage

Sirko Höer

Master Thesis

22th January 2020

Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science
Methods in Multi-Layer Usable Security Research

Design, Implementation and Evaluation of State dependent Fuzzing
Master Thesis

Submitted by Sirko Höer
Matriculation number: 2864536
Date of submission: 22th January 2020
Version: 1.06

First examiner/Erstgutachter: Prof. Dr. Matthew Smith
Second examiner/Zweitgutachter: Prof. Dr. Michael Meier

Rheinische Friedrich-Wilhelms-Universität Bonn
Institute of Computer Science
Methods in Multi-Layer Usable Security Research

Abstract

In the last years coverage-guided fuzzing has become a more and more popular tool to find security related software failures. There are several solutions on the market like AFL or libFuzzer which can help to employ coverage-guided fuzzing. These generate program inputs with respect to coverage and data-flow information due to compile-time instrumentation. However, modern state-of-the-art approaches of fuzzing utilize random mutation strategies to craft single data packets and do not incorporate internal program state and multi layer application hierarchy.

In this master thesis we present an approach that generates not just one data packet but a chain of data packets depending on the coverage information and a state chain grammar. Using a combination of libFuzzer, Protobuf, the libProtobuf-mutator and our own chain mutation strategies we are able to generate meaningful state chains. Our approach has discovered three new vulnerabilities on large network-based open source projects including an *MQTT-Broker* and an *IDS/IPS - System*. Some of these bugs are potentially exploitable memory access violations.

Zusammenfassung

In den letzten Jahren ist coverage-guided Fuzzing als Testing Methode immer populärer geworden, um sicherheitsrelevante Fehler in Softwareprojekten aufzuspüren. Mit diesem Hilfsmittel ist es beispielsweise *Google* gelungen, im *Chrome Browser* über 16000 Fehler aufzudecken. Populäre Vertreter dieser Testing Methode sind unter anderem AFL (American fuzzy lop) und libFuzzer. Die Grundidee von coverage guided Fuzzing ist das Erzeugen von Programmeingaben auf Basis von Programmfluss- sowie Datenflussinformationen der sogenannten coverage sanitizer.

Allerdings erzeugt diese Methode immer nur einzelne Datenpakete für jede Testiteration. Die Folge ist, dass Zustände in Programmen, die aufeinander aufbauen, nicht erreicht werden können, da nach jeder Testiteration der Programmzustand zurückgesetzt wird.

In dieser Masterarbeit wird ein neuer Ansatz vorgestellt, welcher, in Abhängigkeit von einer gewählten Grammatik, Datenketten erzeugt und der an das zu testende Programm übergeben werden. Durch eine Kombination aus libFuzzer, Protobuf, dem libProtobuf-mutator, und neuer Mutationsstrategien für Datenketten, können nun sinnvolle zustandsabhängige Eingaben generiert werden. Das ermöglicht eine höhere Genauigkeit bei den Tests und hilft tieferliegende Fehler in Softwareprojekten aufzudecken. Es sind bereits 3 neue Software Schwachstellen in netzwerkbasierten Open Source Projekten mit dieser Methode gefunden worden. Sie zeigt damit die Effektivität im Vergleich zu herkömmlichen Fuzzern.

Erklärung zur Master - Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn, 22th January 2020

Place, Date

Sirko Höer

Acknowledgements

Herewith I would like to thank everyone who supported me during the preparation of my Master thesis.
My special thanks go to:

- Prof. Dr. Matthew Smith and Sergej Dechand for the provision of the topic and for the pleasant support
- Stephan Plöger for the constant support, the many helpful tips and the patient and uncomplicated answers to my questions
- To all employees of Code Intelligence GmbH for the pleasant working atmosphere and mutual support
- My proofreaders Thorbjörn Schulz, Michael Steglich, Simon Resch and Sebastian Pöplau for their conscientious error reading and useful suggestions
- My partner Marianne and my parents for their constant support in any form during my studies.

Contents

1 Introduction

In the last decade vulnerabilities in large software projects have become the leading causes for many security incidents for instance data leaks, denial of service attacks or local privilege escalation. [18] Finding security related software failures is an important issue in the software developing process. *Fuzzing* is a powerful testing technology that helps to find such bugs in software projects effectively. Since its introduction in the year 1989 [6] fuzzing has improved in many ways. Starting by generating simple random inputs fuzzers have enhanced the way how they craft the inputs by intelligent mutation strategies and genetic algorithms. After the introduction of *sanitizers* like *coverage-sanitizer* or *address-sanitizer* in the year 2016 [28] [21] a new kind of fuzzing has emerged, named **coverage-guided fuzzing** also known as greybox fuzzing. [22] With the feedback of the coverage sanitizers the fuzzer can track which inputs have reached which code sections. Depending on this information, the fuzzer decides which mutation strategy has to be used to change the next input. Popular fuzzer engines are *AFL* and *libFuzzer*.

Unfortunatly this fuzzing methodology covers only one layer of state depth because for every new fuzzing iteration the fuzzer generates a new single data packet and does not incorporate about the internal state of the system under test. Large projects with the focus on complex network protocols require internal states such as an *FTP-Server* or *MQTT-Broker*. As a result common fuzzers are not able to test deeper layers of such programs because of the state dependencies which might leave more complex vulnerabilities unnoticed. To illustrate this we consider the following problem by an example (File 1.1). Imagine we develop a simple network based program for publishing data on a server. The usual behaviour of the server is that first the client has to connect to the service by the command **CONN**. Subsequently, the client can send the command **PUB** and publish some data to the server. If the length of the data is above the *MAX_BUFFER_SIZE* the server will crash caused by a buffer-overflow.

Fuzzers like *libFuzzer* or *AFL* will never reach the code segments in lines 11-13 because the fuzzers do not incorporate about internal states and thus fail due to the IF statement in line 9. This kind of state dependencies are often used in the field of software development for network-based projects.

```c
char BUFFER[MAX_BUFFER_SIZE];
int state = 0;

int connectionHandle(char *input,int len){
  char * command = getCommand(input);
  if(strncmp("CONN",command,strlen("CONN")) == 0 && state == 0){
    state=1;
      ...
  } else if((strncmp("PUB",command,strlen("PUB")) == 0) && state == 1){
    ...
    state=2;
    char * data = getData(input);
    strcpy(BUFFER,data); // <--- BUG ----
    ...
  }
}
```

File 1.1: Example of a state dependent code snippet

This master thesis we propose a new approach of stateful fuzzing. This is a shared library that provides mutation algorithms and allows the security tester to create a new type of fuzz-target by using the new interface. This library tries to generate a chain of data packets with respect to their grammar. In the backend we use the libFuzzer libraries as well as libProtobuf-mutator. Moreover, we decided to focus on network protocols such as *HTTP*, *DNS* or *MQTT*. With this approach it is now possible to send multiple packets to the system under test depending on the internal state. Thus deeper code paths in the system under test are achieved.

To make reliable statements about the evaluation, we decided to test five real world open source projects multiple times and we were able to identify new unknown vulnerabilities.

1.1 Challenges

The overall goal is to create a library which makes it possible to test applications with several layers of depths. The core idea is the creation of a data chain. One of our challenges was to identify possibilities to create the chain and corresponding mutation algorithms. Mutation strategies of common fuzzers are already very well-developed. However, we need to think about different types of modifications to develop sophisticated mutation algorithms for a data chain. We also have the following requirements for our solution: First we want to minimize the dependencies. Only our library should be included to use the features of state dependent fuzzing. Furthermore, we focus on low overhead and memory footprint to avoid performance weaknesses.

1.2 Outline

The master thesis is organized as follows: First, in chapter 2 we present the current state of academic research and related work. Then in chapter 3 we discuss the fundamental background of fuzzing as well as terms and definitions to understand the following chapter 4. The chapter 5 shows the evaluation of our approach. Chapter 6 concludes the thesis and discusses future work.

2 Related work

Security testing in large software projects is performed by dynamic tests or by static code analysis. However, fuzzing has been long used in areas not directly related to security. In the past it was primarily used to test projects in terms of reliability and fault tolerance. The first and simplest method was developed by Miller et al. to test UNIX utilities by giving them random inputs. [6] This approach is very simple and has its limits when it comes to testing network protocols. Network protocols are in most of the cases state dependent. One example of a simple state dependent fuzzer was WSDigger. It is an open source tool for black-box-testing of form-based web services. [30] Other implementations such as SPIKE [3] and PROTOS [17] have the focus to test specific low-level network protocols.

In the year 2006 SNOOZE was introduced by Greg Banks et al. SNOOZE stands for *Toward a Stateful NetwOrk prOtocol fuzZEr* and is a tool for building flexible, security-oriented, network protocol **black-box** fuzzers. SNOOZE is an extensible tool and consists of the *Fault Injector*, the *Traffic Generator*, the *Specification Parser*, the *Interpreter*, the *State Machine Engine*, and the *Monitor*. Responsible for running the fuzz-tests is the interpreter. It generates inputs with respect to the used protocol specification, a set of user-defined fuzzing scenarios and a module that is implementing scenario primitives. SNOOZE has implemented several protocols such as *URL-Request* or the *SIP*-Protocol. Reasoned by the lack of direct feedback of the system under test, it is difficult to evaluate the fuzzer's performance. However, this fuzzer is capable of building simple, general-purpose input chains with respect to their configurations. It supports the creation of sophisticated fuzzing scenarios that are able to expose real-world bugs. [5]

Another approach to state-dependent fuzzing is *RESTler*, the first stateful REST-API fuzzer. RESTler analyzes the API specification of a cloud service and creates a sequence of requests. It is also an approach of black-box fuzzing. RESTler was developed 2019 by Vaggelis Atlidakis, Patrice Godefroid, and Maria Polishchuk. RESTler generates test sequences by inferring consumer producer dependencies. The main algorithm starts by generating a set of request types and their dependencies. The algorithm determines whether the request sequence is valid by evaluating the return code which is defined as any code in the `200` range. The modul `DEPENDENCIES` checks if all dependencies of a request sequence are satisfied. If all dependencies are satisfied the complete request sequence is rendered by the `RENDER` module. The `EXECUTE` module executes each request in a sequence one by one and each time the module checks the validation of the response. It also stores the values to provide the data later when needed by subsequent requests. Overall RESTler, is a small python script which uses REST-API grammar descriptions to generate HTTP-requests. This approach uses neither instrumentation to observe code coverage nor low-level mutation algorithms. However, RESTler is a great way to test REST-API based web applications with low amount of complexity. [4]

All approaches have one thing in common: They are all black-box fuzzing solutions. Without specific coverage information, it is difficult to determine which input is worth more than another. Also, a black-box fuzzer mutates input for a target program randomly without any knowledge about which parts of the input reaches more code coverage. Coverage-guided fuzzers are much better in this case.

3 Background

In this chapter we describe coverage guided fuzzing in general and the used technology behind it. We show the differences between the approaches of **libFuzzer** and **AFL**. We declare the meaning of coverage sanitizer and explain the details with regard to data and control-flow. In addition, we explain the libraries **libprotobuf** and **libprotobof-mutator**. These are some of the main components of our approach and defines the data structure of the chain as well as the grammar.

3.1 Terms and definitions

First we define and describe all important terms related to this thesis. Basic concepts of computer science are assumed.

Fuzz Testing is a software testing method which generates random data and puts this data into a system under test. It tests the stability and robustness of the system under test.

System under test (SUT) is the definition of a target program which is tested for correct operations for example with fuzzing or other testing methods like unittests. The term *SUT* will be used in this thesis.

Unittest is the usual method to test software. In a unittest the developer defines an input for a function, calls this function and compares the output with the desired output.

Fuzz-Targets are functions or class methods which are tested with the help of a particular fuzzer engine.

Fuzzer Engine is the library or the program which is a part of the fuzzer. There are several fuzzer engines on the market such as **AFL** [31],**libFuzzer** [22] or **honggfuzz** [13].

American fuzzy lop (AFL) is one of the first coverage guided fuzzers. It was developed in the year 2013. AFL is a so called *Out-of-Process-Fuzzer* which means that the fuzzer launches a new process for every test case.

LLVM stands for Low Level Virtual Machine and is a compiler infrastructure. It contains a so-called front end for programming languages for example *c*, *c++*, or *objective-c* as well as a back end for any instruction set architecture. LLVM also provides several tools for static code analysis or code coverage reports.

libFuzzer is a part of the compiler runtime (*compiler-rt*) of LLVM. libFuzzer is developed to fuzz test libraries and applications. This engine is also called an *In-Process-Fuzzer* which means that the fuzzer does not launch a new process for every test case, and mutates inputs directly in memory.

Sanitizers are a method to provide fault classification and detection. LLVM provides five types of sanitizer: Address, Memory, Thread, Undefined behaviour and DataFlowSanitizer.

Coverage sanitizers are used to measure code coverage of the system. libFuzzer makes use of the coverage information to generate better inputs for the SUT.

3.2 Coverage-based Greybox Fuzzing

Fuzzing is an effective method to test programs for stability and resistance against wrong inputs. We differentiate the fuzzing approaches into three distinct types: **black box**, **white box** and **grey box** fuzzers.

Black box fuzzing is used as a simple dynamic test. A program generates random or mutated inputs based on seed files and puts the generated data to the SUT to test correct operation. The fuzzer observes only the input / output behaviour of the SUT and notify the user if something goes wrong. Black Box testing is also called IO-driven or data-driven testing. Most of the common fuzzers today are in this category. [23]

White box fuzzing is the opposite of Black Box Fuzzing. With the knowledge of the source code the testing method will automatically generate test cases. By analyzing the internals of the SUT a programm will creates program code with specified inputs and run it within the test environment. It was introduced by Godefroid [11] in 2007 and is also closely related to symbolic execution tight. [12]

Grey box fuzzing makes use of the advantages of both sides. On the one hand the fuzzer gets information of the internals of the system under test while the test is running. On the other hand the fuzzer generates and mutates the input based on the information and puts this input to the SUT. Modern fuzzers such as AFL and libFuzzer are examples in this category. [22] [31]

The first *black box* fuzzer was developed in the year 1989 by Miller et al. [6] to understand the reliability of Unix tools. In the year 2013 mutation-based fuzzing was invented. One of the most popular fuzzers which uses mutation-based algorithms is **AFL** [19] and was developed by Michał Zalewski [31]. A mutation-based fuzzer mutates existing test-cases to generate new test-cases [7] without any input grammas or input model specifications. A few years later coverage-guided fuzzing was introduced. Two of the most known coverage-guided fuzzers are American Fuzzy Lop (AFL) and libFuzzer. Furthermore, there are also implementations of hybrid fuzzers such as Driller. Driller uses AFL as the fuzzer engine to test a SUT and make use of symbolic execution to get new inputs if the fuzzer does not reach new code paths. [24]

3.2.1 Fuzzing Engine AFL

American Fuzzy Lop (AFL) is a security-oriented grey box fuzzer. It uses a novel type of compile-time instrumentation and genetic algorithms to discover clean, meaningful test cases that trigger new internal states in the SUT. AFL is an out-of-process fuzzer, that means every new fuzzing iteration will spawn a fork of the SUT. The biggest advantage is that every new iteration resets the internal state of the SUT. So with every new fuzzing iteration the tested programm starts at the same starting point. The following figure 3.1 shows the general workflow of ALF and it's fork-server.

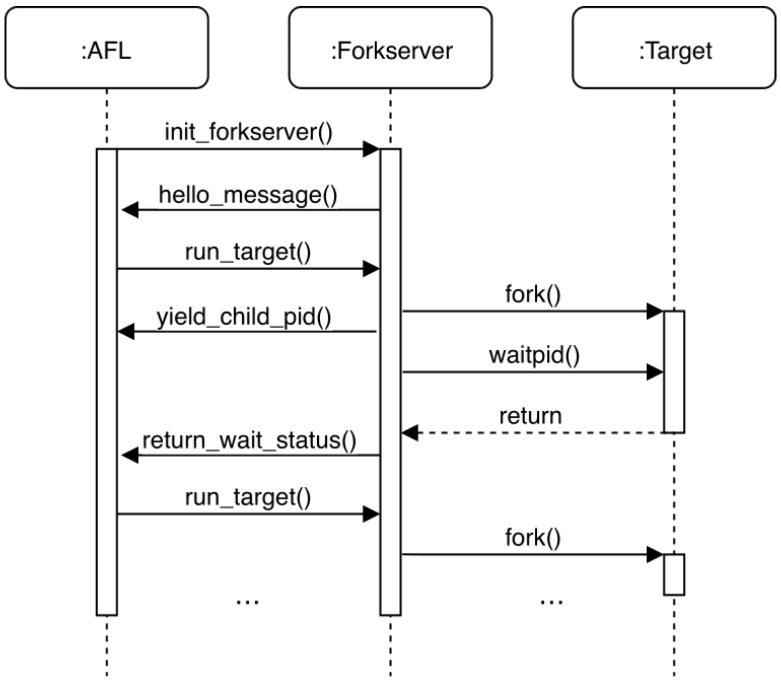


Figure 3.1: Illustration of the fork server of AFL.

As the image shows every iteration forks the target program and waits until the target program finishes. The fork process is time-consuming and AFL can't achieve high performance per seconds. [31] We decided to not use AFL in our fuzzer engine because of the performance weaknesses compare to libFuzzer. It is also much easier to implement an additional library for libFuzzer. We do not want to change the sources of the Fuzzer engine. Instead, we wanted to implement an extension that does not affect the source code of the fuzzer engine.

3.2.2 Fuzzing Engine libFuzzer

LibFuzzer is a part of the compiler runtime (**compiler-rt**) of the LLVM project. The LLVM project includes the LLVM back-end as well as the compiler front-end clang. It supports a cross-compile environment and several tools for analyzing and optimizing source code. [22] The **compiler-rt** contains several sanitizers, such as address, memory or thread sanitizer as well as the coverage sanitizer and the fuzzer engine. The following illustration shows the modules of libFuzzer.

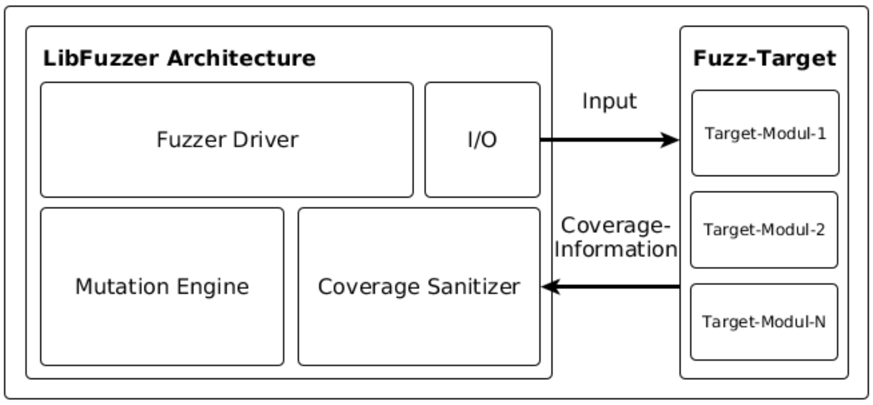


Figure 3.2: Illustration of the architecture of libFuzzer

Fuzzer Driver combines all main modules of libFuzzer. The most important task of the Fuzzer Driver is to provide the main fuzzer loop as well as the option parsing. Also, the Fuzzer Driver is responsible for special task like parallel executing of Fuzz-Targets with forking. [22]

I/O is responsible for internal memory management of libFuzzer. The main task of **I/O** is to provide memory space for the input data and manage complex data structures. [22]

Mutation Engine is one of the key modules of libFuzzer. The mutation engine proceeds as follows: First the mutator starts to mutate a selected input and tests this input against the SUT. Second, if the new input causes an increase in coverage, it is added to the input corpus. LibFuzzer supports a set of mutation operations for instance *EraseBytes*, *ChangeBytes*, *ShuffleBytes*, *CopyPart*, *CrossOver* and others. [8]

Coverage Sanitizer are functions which are called to get coverage information from the target program or library. There are two kinds of coverage sanitizer, control-flow sanitizer and data-flow sanitizer. [22]

In the next section we show details about coverage sanitizer and describe how the technique works.

3.3 Coverage Sanitizer

Coverage sanitizer are a powerful method to measure code coverage. The code coverage is information which is used to craft inputs for the SUT. In addition, the information can also be used to measure the exact code coverage for given Unit- or Fuzztest. Coverage sanitizer are compile time instrumentation. The compiler inserts calls to user-defined functions on function-, basic block-, and edge-levels. There are two fundamental types of coverage sanitizer, the **control-flow-** and **data-flow** coverage sanitizer. [28] [29]

3.3.1 Control-flow coverage sanitizer

To get coverage information of the target program the compiler has to insert callback functions during the compile process. These functions are called if the target program reaches specific parts or sections. The first implementation of the callback functions are called *trace_pc_guard*. These callback functions are tracking the control-flow with different accuracies. There are three kinds of levels, the *function-level*, the *basic-block-level* and the *edge-level*. The user has to define the level of accuracies by setting the corresponding options. [21]

function-level The compiler inserts the callback function at any function of the SUT.

basic block level The compiler inserts the callback function at every basic block of the SUT. A basic block is a code sequence with no branches in except to the entry and no branch out except at the exit.[15]

edge-level The compiler inserts the callback function at any edge of the SUT. The edge level provides the most detailed coverage information. [2]

In the last years the LLVM-project introduced a new kind of control-flow coverage sanitizer, namely the *Inline 8bit-counter*. The difference between *trace_pc_guard* and *Inline 8bit-counter* is instead of inserting a callback function they simply increments a counter. This approach of incrementing a counter is much faster than calling functions. [21]

3.3.2 Data-flow coverage sanitizer

Data-flow sanitizers try to track the data-flow of the target program and stores values of the data-flow into a temporary dictionary. This kind of sanitizer supports the data-flow-guided fuzzing in such a way that the fuzzer generates the input depending on the sanitizer's information. In the following we describe the main data-flow sanitizer. The compiler will insert the data-flow sanitizer around comparison instructions and switch statements as well as data arrays. The data-flow sanitizer helps libFuzzer to generate dictionaries during the fuzzing process. LibFuzzer provides two generated dictionaries: One of them is a temporary dictionary and stores every part of values that were found using the data-flow sanitizer. The other one is a permanent dictionary which stores values if an entry from the temporary dictionary causes good coverage result. The following table 3.1 describes all current types of data-flow sanitizer. [28] [21]

Type	Description
sanitizer_cov_trace_cmp1/2/4/8(...);	This callback function traces the comparisons between two data fields with a size of 1, 2, 4 and 8 bytes.
sanitizer_cov_trace_const_cmp1/2/4/8(...);	This callback function traces the comparisons between a data field and a constant value with a size of 1, 2, 4 and 8 bytes.
sanitizer_cov_trace_switch(...);	This callback function traces the switch statement and each case.
sanitizer_cov_trace_div4/8(...);	This callback function traces division instructions.
sanitizer_cov_trace_gep(...);	This callback function traces the special LLVM Gep (GetElementPtr) instructions to capture array indices.

Table 3.1: Description of different types of data-flow sanitizer

3.4 Protobuf and libProtobuf-Mutation

Proto Buffers (*Protobuf*) is a library that provides a language-neutral, platform-neutral, extensible mechanism for serializing structured data. Initially Protobuf has been developed by Google to solve the problem of large number of requests and responses to a server. It supports *C++*, *Java*, *go* and many other programming languages.

The developer defines a so called *protobuf*-description. The proto-files are translated into native language code. The native language code provides wrapper classes for reading and writing message content. The following file 3.1 is a description of a proto-file. [14] [26]

```
/*                           Protobuf Example File                    */
syntax = "proto3";
package Person;

/* example message */
message Data {
  enum GenderType {
    MAN = 0;
    WOMAN = 1;
  }
  GenderType gender = 1;
  string name = 2;
  string first_name = 3;
  int32 age = 4;
}
```

File 3.1: Example of a protobuf description file

In this example we defined a *message* **Data** which contains two strings, an integer as well as an enumeration type. Protobuf supports many primitive data types such as *int32, int64, float, double, fixed32* and others.

In addition to protobuf Google has developed libprotobuf-mutator. Libprotobuf-mutator is a library for randomly mutating protobuf messages. It can be used together with guided fuzzing engines, such as libFuzzer. Libprotobuf-mutator is used to mutate server request for example *HTTP-requests* or *DNS-packets*. [16] [29]

Details of the definition of the proto-files are detailed described in the section 4.1.2.

4 Methodology

Usually, a fuzzer generates a single data field and the length of the data field differs in each iteration. Depending on the information provided by the coverage instrumentation the fuzzer adds bytes, deletes bytes or mutates the given data values. This data field is used by the system under test (SUT) as an input. Usually, after each fuzzing iteration the state of the SUT will be reset and the fuzzer starts at the beginning. For state dependent fuzzing we need a data structure which contains more than one data field so that we can generate data depending on the state of the system under test. It is a vector of data fields which we called **state chain**.

This chapter describes the design of state dependent fuzzing as well as the essential parts of the implementation. Furthermore, we specify the user interface, each grammar and the mutation algorithms. Finally, we focus on the general workflow as well as the mutation strategies.

4.1 Concept of stateful fuzzing

The stateful fuzzing library is divided into three main components. The first part is a user friendly API which provides a low amount of specification and dependencies. It is a predefined **macro** definition and provides a data type **state_chain**, the size of the **state_chain** as well as the configuration of the state chain. The second part covers the implementation of the grammar components which includes definitions of specific input structures. It also includes the component which converts the protobuf message into the state chain data structure. The third part contains the mutation strategies which are responsible for mutating the data fields within the state chain or the state chain itself. We design three levels of mutation strategies which will be explained in the section 4.1.3.

4.1.1 Design of the concept

As we described, our approach generates a chain of data fields depending on the coverage information of the SUT and the grammar configuration. In the following, we show the workflow of the approach of stateful fuzzing in general.

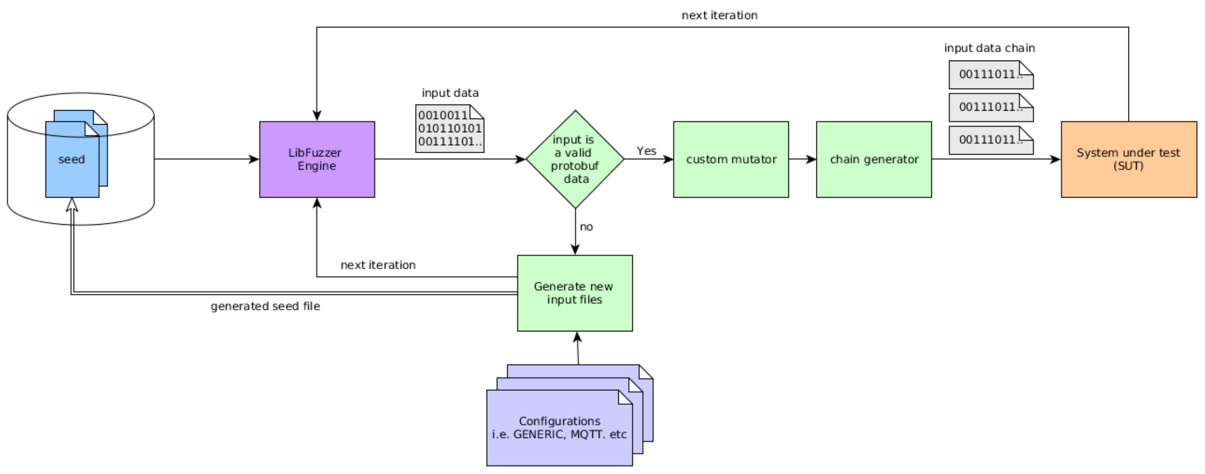


Figure 4.1: Illustration of the workflow of the approach of stateful fuzzing.

From the workflow perspective the process starts with the creation of a data packet, which is assembled by libFuzzer. Depending on the **seed** files the fuzzer will generate either a valid **protobuf** message or a random data packet. If the data packet is not a valid **protobuf** message the algorithm will create a new seed file which contains a valid **protobuf** message based on the given configuration and stores it in the seed folder. This is important for further processing of the data package. After generating a new seed file the fuzzing loop will load the valid file and operate mutations on it. The library provides different mutation algorithms which work on three hierarchical levels. Details about the mutation algorithms are described in the section 4.4. After mutating, the input will be converted from protobuf message into a `vector` of data fields which is delivered to the **fuzz-target**.

Because of several variations of different network protocols we decided to implement a chain configuration. The chain configuration differs in four types, which define how the packet chain looks like. The user has to specify the configuration as well as the minimum and the maximum chain size. The reason to set the minimum and maximum is that some network protocols such as *MQTT* has a minimum set of states for instance at least one `CONNECT` and one `DISCONNECT` packet. It also reduces the complexity. The following enumeration shows the configuration types.

GENERIC CONFIGURATION is a generated state chain based on mutation information. The fuzzing process will start with an empty chain. Each iteration the fuzzer changes the chain by adding, removing or modifying one or more datapackets.

HTTP CONFIGURATION The Stateful Fuzzer generates a grammar that provides an HTTP request. The grammar is described by a **protobuf** message will be converted to a HTTP-request.

DNS CONFIGURATION The Stateful Fuzzer generates a grammar that provides a DNS request and response. The grammar is described by a **protobuf** message will be converted to a DNS data packet.

MQTT CONFIGURATION The Stateful Fuzzer generates a grammar that provides a MQTT message. The grammar is described by a **protobuf** message will be converted to a MQTT data packet.

4.1.2 Definition of the protobuf message for the state chain

The definition of the state chain contains a general protobuf message *ProtoStateChain* which includes a chain type and the chain itself. The chain is represented as a repeated field of *FuzzData*. *FuzzData* is again a protobuf message which contains one of the following messages.

- Generic
- HTTPRequest
- DNSChain
- MQTTChain

The design of the definition is chosen so that it can be easily extended. The following listing shows the complete definition of the protobuf message.

```
/*                              Protobuf State Fuzzer                              */
/* Generic state chain */
message Generic {
  bytes raw_data = 1;
}

/* Choose configurations */
message FuzzData {
  oneof fuzz_data_oneof{
    Generic generic_fuzz_data = 1;
    HTTPPacket.HTTPRequest httpreq_fuzz_data = 2;
    DNSPacket.DNSChain dns_fuzz_data = 3;
    MQTTPacket.MQTTChain ssl_fuzz_data = 4;
  }
}

/* final state chain */
message ProtoStateChain {
  enum ChainType {
    GENERIC = 0;
    HTTP_REQUEST = 1;
    DNS_PACKETS = 2;
    MQTT_PACKETS = 3;
  }
  ChainType types = 1;
  repeated FuzzData fuzz_data = 2;
}
```

File 4.1: protobuf definition of generic state chain

4.1.3 Mutation strategies

The mutation strategy of state dependent fuzzing has a three levels hierarchy. Level one includes the mutation strategies for the state chain. Level two includes the mutation of specific protobuf messages which is performed by the libProtobuf-mutator. Level three is the fall back mutation strategy for generic data field which is performed by the libFuzzer mutation engine.

Level one

Level one is the highest tier of our mutation strategies and mutates the structure of the chain. We designed five varied kinds of mutation options. All of those options have their own probability of occurrence. The following table 4.1 describes all mutation options in detail.

Mutator	**Description**	**Occurrence probability**
generate new chain	Resets the given chain to the state which is defined by the configuration	0.5 %
mutate entries of the chain	Mutates all entries in the chain. Depending on the configuration the entries are mutated either by the libProtobuf-mutator (level two) or by the libFuzzer-mutator (level three)	75 %
swap two data field	Swaps two entries of the chain if the chain contains two or more entries. The entries which will be swapped are selected uniformly	10 %
add new data field	Adds a new entry at the end of the chain. Depending of the configuration the method selects either a generic data field or a grammar based data field.	15 %
remove a data field	Removes a data field uniformly at random.	4.5 %

Table 4.1: Description of the mutation methods at level one

The probabilities have been chosen by empirical research which has shown that this probability distribution has generated the best results. Most of the time (75%) we mutate the data fields within the state chain. New data fields are added with a probability of 15% and tow different data fields are swapped 10% of the time. We decided to set a higher probability of adding new data fields than removing data fields. We have noticed during testing that this more likely to increases the coverage.

Level two

This level of granularity is responsible for mutating single data fields of the state chain. Depending on the configuration type the data field is mutated by libProtobuf-mutator. LibProtobuf-mutator includes several methods to change the values with respect to the coverage information. The following table 4.2 describes all mutation methods. [16]

Mutator	**Description**
Add a new data field	Creates a new data field with respect to the given message description
Mutate a data field	Mutates a part or a field or a protobuf message with respect to their description
Delete a data field	Deletes a single data field or a part of a protobuf message with respect to their description
Copy a data field	Returns a part of an instance of a protobuf message to the input

Table 4.2: Description of the mutation methods at level two [16]

Level three

The third level of granularity is the *fallback* mutation strategy. This strategy is used if level one and two do not have an effect of increase the code coverage. The mutation methods are provided by the libFuzzer library. The random mutation is the reason for a powerful local search to get useful inputs. However, the tests need many iterations to obtain additional coverage. The following table 4.3 shortly describes all mutation operations provided by libFuzzer. [8]

Mutator	**Description**
EraseByte	Reduces size by removing a random byte
InsertByte	Increases size by one random byte
InsertRepeated Bytes	Increases size by adding at least three random bytes
ChangeBit	Flips a Random bit
ChangeByte	Replaces byte with a random one
ShuffleByte	Randomly rearrange input bytes
ChangeASCII Integer	Finds ASCII integer in data, perform random math operations and overwrite into input
ChangeBinary Integer	Finds Binary integer in data, perform random math operations and overwrite into input
CopyPart	Returns part of the input
CrossOver	Recombines with random part of corpus
AddWordPersist AutoDict	Replaces part of the input with one that previously increased coverage
AddWordTemp AutoDict	Replaces part of the input with one that recently increased coverage
AddWord FromTORC	Replaces part of the input with recently performed comparison

Table 4.3: Description of the mutation methods at level three [8]

4.2 Userinterface

The user interface is a *C/C++* `#DEFINE` declaration and is defined in the file `custom_mutators.h` which has to be included to the *Fuzz-Target*. The signature of the interface is defined by
`DEFINE_STATE_CHAIN_FUZZER(state_chain *chain, size_t chain_size, &c_config)`
The first argument of the signature is a value of type `state_chain` which is a `vector` of `pair<string,size_t>`. The entry of type `pair<string,size_t>` contains the input buffer from a data field as well as the size of the data field. The second input parameter is the chain size of the type `size_t`. The last parameter of the declaration `DEFINE_STATE_CHAIN_FUZZER(...)` specifies the configutation of the chain. The following listing is an example (File 4.2) of how a developer can use the interface.

```
#include <vector>
#include "custom_mutators.h"

chain_configuration c_config = {
    SF_GENERIC_CONFIG,              // Type of the state chain (grammar type)
    2,                              // Minimum size of the state chain
    5                               // Maximum size of the state chain
};

DEFINE_STATE_CHAIN_FUZZER(state_chain *chain, size_t chain_size, &c_config) {
 //setup code
 ...

  for (state_chain::iterator item = chain->begin();
    item != chain->end(); ++item){
    string buffer = item->first;
    size_t size = item->second;
    // fuzz this function
    CustomAPI(buffer.c_str(),size);
  }

  // teardown code
  ...
  return 0;
}
```

File 4.2: Example how to use the interface

Definition of the configuration

The third parameter of the `DEFINE_STATE_CHAIN_FUZZER(...)` declaration specifies the attributes of the state chain. The parameter is a data type which describes a struct with the following elements. The first element is a type definition which is a reference to a number. At this point we set four definitions:

- `SF_GENERIC_CONFIG`

- `SF_HTTP_REQUEST_CONFIG`
- `SF_DNS_CONFIG`
- `SF_MQTT_CONFIG`

The example file 4.3 shows at line 2 an instance of the data structure for the configuration. In the following we describe shortly how we implemented each state chain type.

```
1 ...
2 chain_configuration c_config = {
3     SF_GENERIC_CONFIG,              // Type of the state chain (grammar type)
4     2,                              // Minimum size of the state chain
5     5                               // Maximum size of the state chain
6 };
7 ...
```

File 4.3: Example of the struct to define the state chain

4.3 Types of configurations

Each configuration type is referred as a number which is defined in file `custom_mutators.h`. As we mentioned there are four configuration types available. The following section describes all types in detail.

4.3.1 `SF_GENERIC_CONFIG`

The `GENERIC_CONFIG` is the simplest approach of the state chain and consists of a repetition of data fields. The type of the data fields is `string`. The values within the data field are mutated by libFuzzer-mutator or libprotobuf-mutator depending on the mutation level. The `generic state chain` is defined in the file 4.1 at line 8.

4.3.2 `SF_HTTP_REQUEST_CONFIG`

The `HTTP_REQUEST_CONFIG` consists of an HTTP request with several attributes. Usually an HTTP request includes attributes for example the **method**, the **URL**-felt, the **HTTP-version**, the **host** and many more. For simplicity, we decided to restrict the amount of attributes. The following protobuf description shows all selected attributes. The `http request state chain` is defined in the file 4.4.

```
/* Protobuf State Fuzzer */
/* http_request header */
message HTTPRequest {
  /*GET /tutorials/other/ HTTP/1.1 */
  Url request_line = 1;
  /*Host: net.example.com */
  Host host_line = 2;
  /*User-Agent: Mozilla/5.0 (Windows; U; Windows ... */
  UserAgent user_agent_line = 3;
  /*Accept: text/html,application/xhtml+xml,application/xml;q=0.8 */
  Accept accept_line = 4;
  /*Accept-Language: en-us,en;q=0.5 */
  AcceptLanguage accept_language_line = 5;
  /*Accept-Encoding: gzip,deflate */
  AcceptEncoding accept_encoding_line = 6;
  /*Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7 */
  AcceptCharSet accept_charset_line = 7;
  /*Keep-Alive: 300 */
  KeepAlive keep_alive_line = 8;
  ConnectionControl connection_line = 9;
  Cookie cookie_line = 10;
  Pragma pragma_line = 11;
  CacheControl cache_control_line = 12;
}
```

File 4.4: protobuf definition of the HTTP request

First we define the url request line which includes the *method*, the *server path* and the *HTTP version* followed by the *host*-line which usually includes the hostname of the web-server. After that comes the *UserAgent*, the Accept attibutes, *Keep-Alive*-line, and *Connection* followed by *Cookie*, *Pragma*, and *Cache-Control* which are optional.

4.3.3 `SF_DNS_CONFIG`

The DNS protocol is well-documented in the RFC 1035 [1]. DNS is a hierarchical client-server protocol. Each domain is served by one or more DNS servers. Usually a client sends a request to the DNS server to get a certain IP address from a given fully qualified domain name (FQDN). However, a typical *DNS-request* includes a chain of packets namely a ***Question***, an ***Answer***, an optional ***Authority***, and an optional ***Additional*** packet as well. All packets follow the same ***DNS-Header*** format. Those DNS request chain is described in the protobuf message below. (File 4.5)

```
/* Protobuf State Fuzzer */
...
message DNSChain {
  /* packet header of a dns request => */
  DNSHeader dns_header = 1;
  oneof dns_packets_oneof{
    /* 1. packet of a dns request (question) => */
    DNSQuestion dns_question = 1;
    /* 2. packet of a dns request (awnser)  <= */
    DNSAnswer dns_answer = 2;
    /* 3. packet of a dns request (auth request)  <=> */
    DNSAuthority dns_authority = 3;
    /* 4. packet of a dns request (optional)  => */
    DNSAdditional dns_additional = 4;
    /* Vanilla byte field */
    bytes dns_vanilla = 5;
  }
}
```

File 4.5: protobuf definition of the DNS packets

Each DNS packet consists of the DNS header and in addition the *Question*, the *Answer*, the optional *Authority* or the optional *Additional*.

DNS-Header includes an *ID*, *`OPCODE`*, several flags, a *`QDCOUNT`*, an *`ANCOUNT`*, a *`NSCOUNT`* as well as an *`ARCOUNT`*. As we mentioned each DNS-packet follows the same ***DNS-Header*** as well as the *`QUESTION`* or *`ANSWER`*. A DNS question includes three fields namely a *`QNAME`*, a *`QTYPE`* and a *`QCLASS`*. The *DNS-Anwser* includes particular data fields namely a *`NAME`*, a *`TYPE`*, a *`CLASS`*, the *`TTL`*, an *`RDLENGTH`* and an *`RDATA`* field.

4.3.4 SF_MQTT_CONFIG

The `MQTT` (MQ Telemetry transport) protocol is a Client Server publish/subscribe messaging transport protocol. MQTT runs over TCP (Transmission Control Protocol) or other network protocols that includes ordered, lossless, bi-directional connections. It is light weight, simple and designed to be easy to implement. MQTT is a state dependent network protocol and first needs a connection establishment. If the state connected is reached, the client can send certain messages of types like

- `publish`
- `subscribe`
- `unsubscribe`
- `ping request`

and many more. All MQTT message types are described at the protobuf message below.

```
/* Protobuf State Fuzzer */
...
message MQTTChainBody {
  oneof mqtt_packets_oneof{
    MQTTConnAck   mqtt_conn_ack       = 1;
    MQTTPub       mqtt_pub            = 2;
    MQTTPubAck    mqtt_pub_ack        = 3;
    MQTTPubRel    mqtt_pub_rel        = 4;
    MQTTPubRec    mqtt_pub_rec        = 5;
    MQTTPubComp   mqtt_pub_comp       = 6;
    MQTTSub       mqtt_sub            = 7;
    MQTTSubAck    mqtt_sub_ack        = 8;
    MQTTUnSub     mqtt_unsub          = 9;
    MQTTUnSubAck  mqtt_unsub_ack      = 10;
    MQTTPingReq   mqtt_ping_req       = 11;
    MQTTPingResp  mqtt_ping_resp      = 12;
    bytes         mqtt_vanilla        = 13;
  }
}

message MQTTChain {
  MQTTConn mqtt_conn = 1;
  repeated MQTTChainBody = 2;
  MQTTDisconn mqtt_disconn = 3;
}
```

File 4.6: protobuf definition of the MQTT packet chain

Our description of the protobuf message (File 4.6) considers the fact that every connection to a MQTT server (also called MQTT-Broker) starts with the `connection packet`. After connecting, our implementation will permute all other MQTT messages except the MQTT disconnection message. This design strategy includes not only the usual behaviour of the protocol. It also includes MQTT messages that are normally sent by the MQTT-Broker to trigger bugs or undefined behaviour within the MQTT-Broker or MQTT-Client.

4.4 Mutation algorithms

In section 4.1.3 we described the hierarchy of mutation strategies in three levels. The first level mutates the structure of the chain. This level implements five mutation operations like **generate new chain**, **mutate entry**, **swap**, **add new entry** and **remove one entry**. Each of the mutation operations are picked by our distribution and has been chosen by empirical research. The **Algorithm** 1 illustrate the behaviour how the algorithm on the level one works. We get as an

Algorithmus 1 : Choose mutation operation

```
Data : message, seed, length, max_length
Result : choose a mutation operation by a uniform random distribution
Let message_number_array ← ∅ ;
Let mutation_op ← 0 ;
Let random ← NewRandomEngine(seed);
Let mutator ← LibProtobufMutator(random);
while not at end of the protobuf message do
    Let description ← GetDescription(message);
    if HasMessage(description) then
        message_number_array ← message_number;
end
if message_number_array is ∅ then
    mutation_op ← generate_new_chain;
else
    mutation_op ← getRandomMutationStrategy();
end
switch mutation_op do
    case generate_new_chain do
        GenerateNewChain(message);
    end
    case mutate_chain do
        MutateGivenChain(message,max_length,mutator);
    end
    case swap_entries do
        SwapPacket(message,random);
    end
    case add_packet_to_chain do
        AddEntryToChain(message,max_length,mutator);
    end
    case remove_packet_from_chain do
        RemoveEntryFromChain(message,max_length,mutator);
    end
    otherwise do
        do nothing
    end
end
```

input a protobuf message, a seed number, a length of the message as well as the maximum length. First we create a set of message numbers to store all message numbers within the protobuf message. Also, we create a local variable to store the chosen mutation operation. Thereafter, we iterate over the message to get the message numbers from all entries of the state chain. Then we check if the array is empty. In case the array is empty the value *generate_new_chain* will be assigned to variable mutation_op otherwise we get a value chosen by *getRandomMutationStrategy()*. In table 4.1 we present the exact occurrence probability distribution. The last step is a case distinction with respect to the value of *mutation_op*.

The mutation operation **mutate entry** uses existing mutation algorithms. In this case we call it mutation level two and three. The mutation level two uses the libprotobuf-mutator that changes the data field depending on its protobuf description. Level three uses the libFuzzer-mutator to mutate the data field on a binary level.

4.4.1 Generate new chain

The task of the algorithm is to create a new and valid state chain depending on given chain configuration and is called with a likelihood of 0.5%. It does not matter whether the given state chain contains entries or not. The **Algorithm** 2 describes the algorithm.

Algorithmus 2 : Generate new state chain

```
Data : message, max_length, mutator
Result : generate an new chain within the message
Let message.type ← chain_config;
for $i = 0$; $i < message.data\_size()$; $i++$ do
    switch message.type() do
        case generic do
            llvm_mutate_entry(message.data(i),max_length);
        end
        otherwise do
            libprotobuf_mutate_entry(message,i,mutator)
        end
    end
end
while message.size ¡ min_chain_length do
    message.add_entry();
    switch message.type() do
        case generic do
            message.set_data(message.data( message.chain_size() -1 ).data();
        end
        otherwise do
            Let Grammar ← getGrammarData(chain_config);
            message.set_data(message.data( message.chain_size() -1 ).data(Grammar);
        end
    end
end
```

First we set the internal chain type to the given configuration. Then depending on the type the algorithm chooses either the mutation level two by using *libProtobuf-mutator* on protobuf messages or the mutation level three by using *libFuzzer-mutator* to mutate all existing entries of the chain. If the chain type is *generic* then the algorithm chooses mutation level three, otherwise mutation level two. Afterwards we add new entries until the minimum chain length is reached.

4.4.2 swap two entries

The task of this algorithm is to swap two entries of the chain and is called with a likelihood of 10%. The two entries are chosen uniformly at random. The **Algorithm** 3 describes the algorithm.

Algorithmus 3 : Swap two entries of the state chain

```
Data : message, random_engine
Result : Swap two entries of the state chain within the message
if message.getStateChain.size() leq 2 then
    return;
Let data_chain ← message.getStateChain();
Let index_1 = getIndexByUniformRandomDist(message,random_engine);
Let index_2 = getIndexByUniformRandomDist(message,random_engine);
if index_1 eq index_2 then
    return;
else
    data_chain.swapElements(index_1,index_2);
end
```

First we check if the chain contains at least two elements. Afterwards we collect two indices of the state chain uniformly at random. If the indices are the same we return from our algorithm without swapping otherwise we swap the entries.

4.4.3 add one new entry

The **add** algorithm appends a new entry to the state chain and is called with a likelihood of 10%. The following **Algorithm** 4 describes the behaviour.

The behaviour is similar to the second part of the **Algorithm** 2. First we prove the size of the chain. If the size reaches the maximum chain length the routine will return, otherwise the routine will add a new entry depending on the configuration. After adding the new entry, we mutate this entry as well.

Algorithmus 4 : Add new entry to state chain

```
Data : message, max_length, mutator
Result : add new entry to the state chain within the message
message.add_entry();
if(message.getStateChain.size() lo max_chain_config) return;
switch message.type() do
    case generic do
        message.set_data(message.data( message.chain_size() -1 ).data();
    end
    otherwise do
        Let Grammar ← getGrammarData(chain_config);
        message.set_data(message.data( message.chain_size() -1 ).data(Grammar);
    end
end
MutateGivenChain(message,max_length,mutator);
```

4.4.4 remove one random entry

The **remove** algorithm deletes a random entry out of the state chain and is called with a likelihood of 4.5%. The following **Algorithm** 5 describes the behaviour.

Algorithmus 5 : Remove entry from state chain

```
Data : message
Result : remove an entry from the state chain within the message
Let random_number ← (rand() % message.chain_size());
Let stack_pointer ← 0;
message.getChainData(random_number).Clear();
for $i = 0$; $i < message.data\_size()$; $i++$ do
    if message.getChainData(i).ByteSizeLong() then
        message.getChainData().Swap(i,stack_pointer);
        stack_pointer++;
end
for $rev\_i = (message.chain\_size() - 1)$; $i >= 0$; $i--$ do
    if message.getChainData(i).ByteSizeLong() then
        message.RemoveLast();
end
```

Basically, we get at first a random index between zero and the chain size minus one and store it into a variable `random_number`. We also need a stack pointer which stores the position of the entries of the chain in every iteration. In the next step we clean the data from the entry with the index of the value of `"random_number"`. The first loop rearranges the chain so that the empty entry is at the last position. The second loop removes the empty entries.

5 Evaluation

Now we show that, in practice, our approach of state dependent fuzzing is more effective and gainful than a usual state of the art fuzzer like libFuzzer. To determine this, we perform an evaluation on five open source projects written in `C/C++`. The goal of the evaluation is to compare the found features and code coverage with a usual guided fuzzer. More specifically, we compare our approach with **libFuzzer** (Version 9). Finally, we proudly show our bug findings which are the best evidence that the approach works.

5.1 The setup

We evaluate the stateful fuzzing library on five open source projects, namely the **libupnp library**, **suricata IDS**, **mosquitto MQTT-Broker**, the **openssl library**, and the **lighttpd Webserver**. All projects are related to network communication and state dependencies. We selected all the projects because we expect an increase of the code coverage as well as vulnerabilities such as memory corruptions, undefined behaviour and many more.

All tests are performed on a four core / eight threads **Intel(R) Core(TM) i7-7700HQ CPU 2.80GHz** and 32 GB of RAM.

Each project has at least two scenarios, one performed by the **libFuzzer** fuzzing engine and one performed by the new **stateful fuzzing library**. Both evaluation test scenarios runs either 4 or 8 hours depending on the scope of the project. We have repeated every scenario ten times. We decided to limit the runtime because we evaluate that the increase of code coverage is no longer significant after 4 hours, in large projects such as *OpenSSL* after 8 hours. To avoid huge variations of the test results we start every run with the same seed random number, the magic number. Fuzzing in general is a probability based approach, so we decided to always set the same seed random number and prevents a large variation of the results. In total, we run about 120 tests and more than 240 hours of runtime to get comprehensive meaningful results.

All targets are compiled with following coverage sanitizer. (See table 5.1)

Compile Options	Effect
-fsanitizer=	
fuzzer-no-link	general set of fuzzing instrumentation
address	sanitizer for memory corruptions
undefined	sanitizer for undefined behaviour
signed-integer-overflow	sanitizer for integer overflows
bool	sanitizer for bool abuse
pointer-overflow	sanitizer for pointer-overflow
-fsanitize-coverage=	
trace-cmp	sanitizer to trace and store strings
indirect-calls	sanitizer for indirect calls
inline-8bit-counters	inserts an counter to track edge coverage
pc-table	type of storage of the program counter
trace-div	sanitizer to trace divisions
trace-gep	sanitizer to trace array indexes and their values

Table 5.1: Description of the compiler options of the system under test [22]

5.1.1 Metric

We have decided to use the code coverage over time as our first metric for the evaluation. It is a reliable metric and is used by many scientists [19] . Also, we mesure the found features and the performance to make statements about usability. We compare our approach with libFuzzer as the baseline. In our evaluation we measure the pointwise average, minimum as well as the pointwise maximum. The measuring of the performance in our approach is a difficult task. LibFuzzer logs the performance by counting the executions per seconds with respect to one data packet. However, our approach calls the Test API several times because of the data chain, so it is difficult to compare performance values.

5.1.2 Limitations

All evaluation tests are executed without seed files and dictionaries. We use no parallel fuzzing instances, and we don't run the tests on different systems. The options for the fuzzer runs are always the same. The following table 5.2 describes all options of the fuzzer runs.

Option	**Effect**
-reduce_inputs=1	Reduce size of the corpus data
-use_value_profile=1	The fuzzer will collect value profiles for the parameters of compare instructions and treat some new values as new coverage.
-shrink=1	Reduce size of the corpus data
-focus_function=auto	The fuzzer tries ot create new inputs, to trigger new program path and functions
-reduce_depth=4	Limitation of the depth of reduction of the inputs It reduces the complexityof the computation of new inputs
-rss_limit_mb=32000	Limitation of the allocated memory
-print_pcs=1	Printing new code paths
-print_ticks=1	Print coverage information every 5 seconds

Table 5.2: Description of the options of the fuzzer runs [22]

5.2 The Experiments

5.2.1 libupnp Version 1.8.4

Libupnp is a library and SDK for UPnP devices. It is designed to build control points, devices or bridges that are compliant with Version 1.0 of the Universal Plug and Play Device Architecture Specification. It supports several operating systems like Linux, BSD, Solaris and others. Originally the software was developed by Intel in 2000 and since 2006 maintained by Michel Pfeiffer under open sources licence.

The core task of UPnP is to enable discovery, event notification, and the control of devices on a network. It is based on common internet standards and specifications such as TCP/IP, HTTP, and XML. We evaluate the *mini-web-server* module within the libupnp SDK. The communication protocol is highly state dependent. Control or notifications are requested in a specific order. Every single request changes the internal state of the program. [25]

Evaluation Fuzz-Target

The evaluation test focuses on the raw network data streams. Both, the usual fuzzer and the state dependent fuzzer, work according to the same principle by sending the generated inputs via a socket connection. In the following image 5.1 we show the results of our experiment.

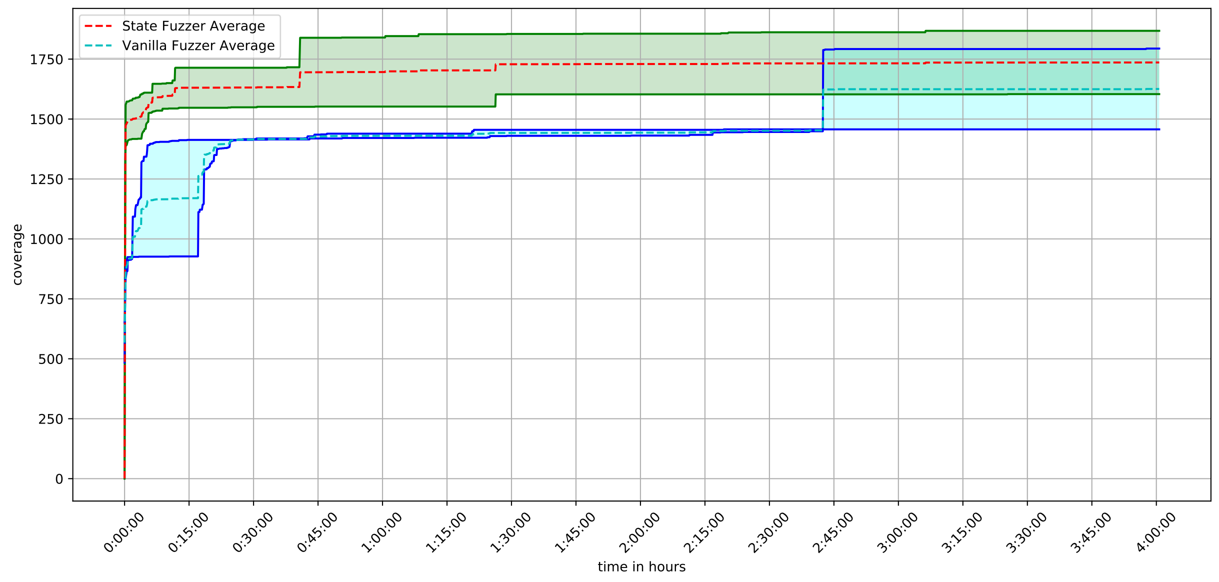
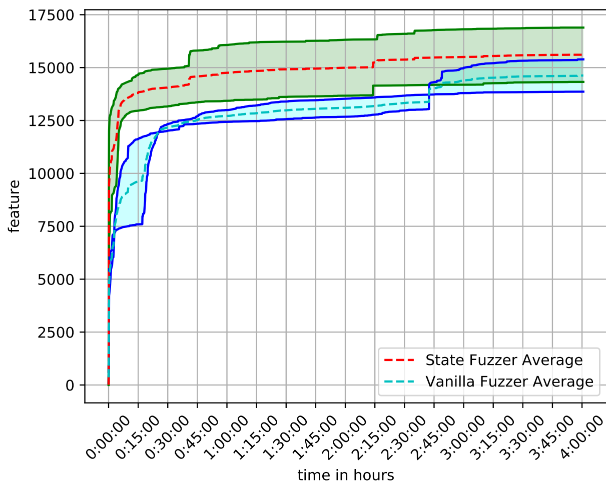
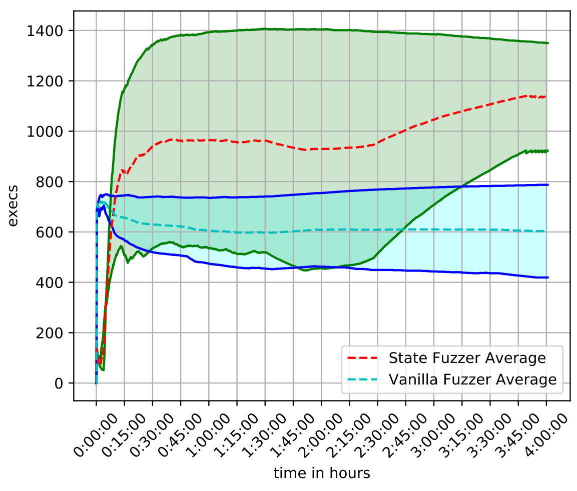


Figure 5.1: Libupnp comparison of the code coverage between libFuzzer and the stateful fuzzer



a) Comparison of found features



b) Comparison of the performance

Figure 5.2: Libupnp comparison of found features and executions per seconds between libFuzzer and the stateful fuzzer

In the results of the first experiment we observe an improvement of code coverage about 15% in the average case, 20% in the best case. The same applies to the found features. More interesting is the huge variation of the execution per seconds. In fact, we improve the execution performance by about 35%. The more time passes in the test, the better the performance of the tests. We don't expect this improvement of the performance. Because it is a network application, the fuzz-target works over socket communications. The operating system restricts socket communication so far that, if connections are established incorrectly, the operating system waits a certain time before releasing the socket. Because the fuzzer assembles a correct data chain, the operating system does not run into a timout. The baseline fuzzer (libFuzzer) has a higher probability of creating an incorrect connection.

5.2.2 Suricata IDS Version 5.0.0

In the next experiment we decided to test suricata. Suricata IDS is an open source real time intrusion detection system (IDS), inline intrusion prevention system (IPS) as well as a network security monitor (NSM) to detect unwanted behaviour in local networks. [10] It was originally introduced in the year 2010. It contains about 500k lines of code distributed over 600 files. It's partly written in C and partly in Rust. Suricata is widely supported by the community as well as by official sponsors like DCSO (Deutsche Cyber-Sicherheitsorganisation GmbH).

For the evaluation our focus was to test the *App-Layer-Parser* as well as the *TCP-Decoder* of Suricata. We decided to create for each component at least two fuzz test because of the complexity of suricata. The evaluation test focuses on generating raw data packets and calling the API functions. [10]

App-Layer-Parser

The *AppLayerParser* is responsible for parsing data within the most frequently used network application protocols like *HTTP, FTP, JABBER, SSH, TLS* and many more. As a matter of fact, we test the function **AppLayerParserParse()**. The following image 5.3 describes the results and clearly shows the advantages of our appoach.

In all of the 10 tests of the *AppLayerParser* which runs 4 and 8 hours we are able to increase the code coverage by nearly 100%. After only about half an hour we increase the reached code regions by 75%. The same applies to the features achieved and can be seen in figure 5.4. The performance of the fuzzer runs depends on the current fuzz-target and is in this case approximately 10% faster than libFuzzer. After 10 instances of testing, our stateful fuzzer is able to achieve deeper coverage with sometimes more than 75% of covered lines (i.e. app-layer-dcerpc.c reached 86% compare to 8% with libFuzzer).

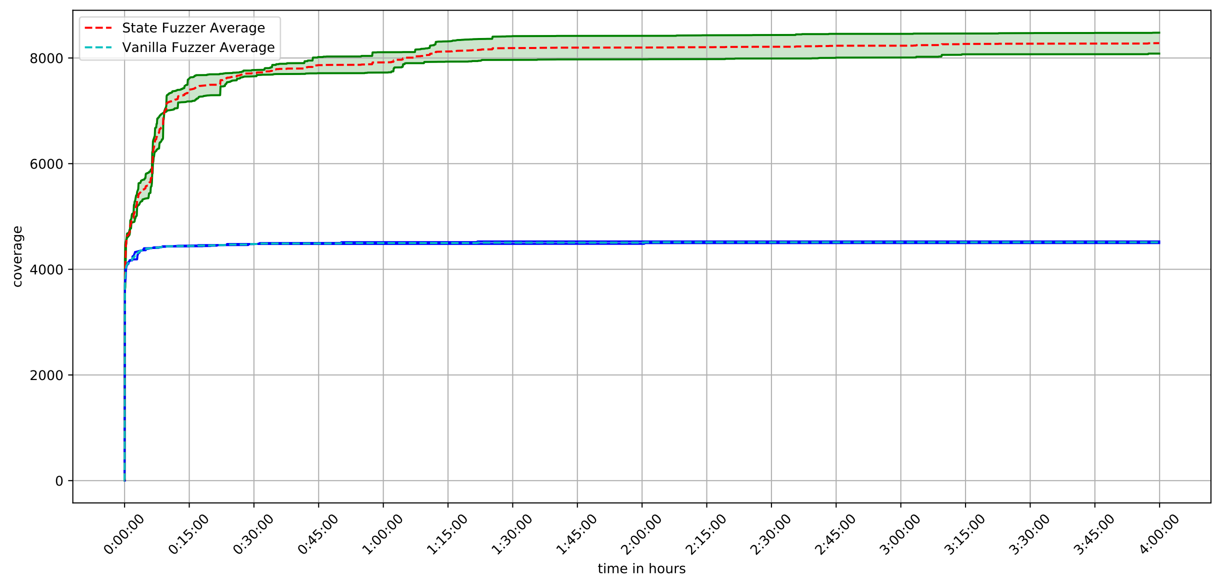
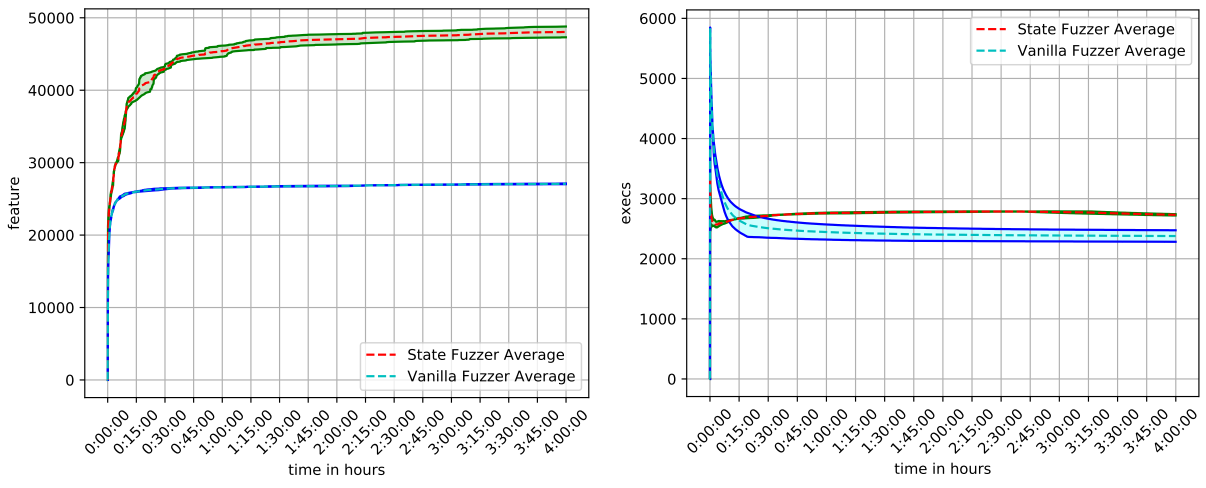


Figure 5.3: Suricata AppLayerParser comparison of the code coverage between libFuzzer and the stateful fuzzer



a) Comparison of found features **b)** Comparison of the performance

Figure 5.4: Suricata AppLayerParser comparison of found features and executions per seconds between libFuzzer and the stateful fuzzer

TCP-Decoder

The *TCP-Decoder* is responsible for parsing raw data of the TCP stack of an internet connection. In fact, we test the function **DecodeTCP()**. The following image set 5.5 and 5.6 describes the results.

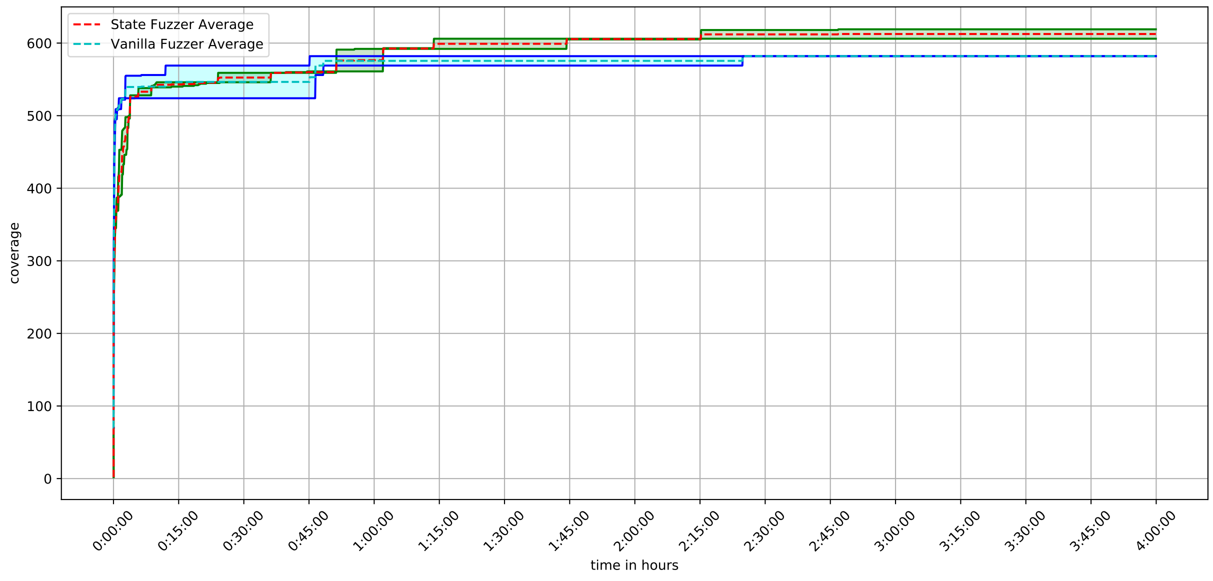
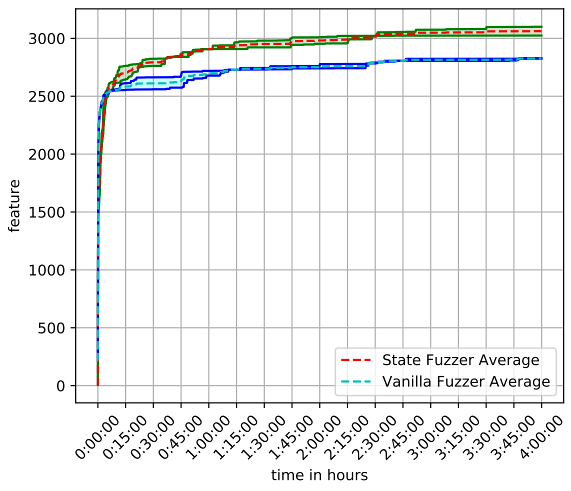
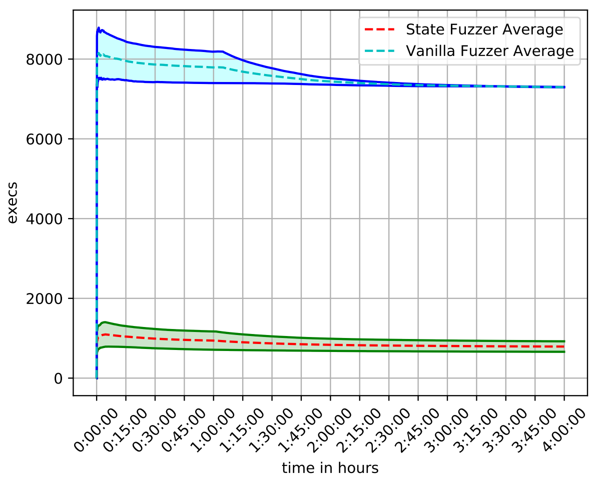


Figure 5.5: Suricata TCP-Decoder comparison of the code coverage between libFuzzer and the stateful fuzzer

In the test using the *TCP-Decoder*, our approach performed a little better than libFuzzer. Therefore, we are able to increase the code coverage by not more than 10%. Only after about one hour we increase the reached code regions by more than 5%. The same applies to the features achieved and can be seen in figure 5.6. The performance of the fuzzer runs in our apporach are much worse than the runs with libFuzzer. In fact, we lose in the test about 80% of the



a) Comparison of found features



b) Comparison of the performance

Figure 5.6: Suricata TCP-Decoder comparison of found features and executions per seconds between libFuzzer and the stateful fuzzer

performance. However, we achieve in the same time at least the same code coverage and in the end about 10% more. In conclusion, we achieve with our approach of state dependent fuzzing at least 10% more code coverage and depending on the application area (in our case the AppLayerParser) we increase the code coverage by nearly 90%.

Findings

The best case to show that our approach works is to publish software vulnerability found by our stateful fuzzer. We found two new vulnerabilities in suricata. The following table 5.3 gives an overview of the bugs found.

Type of vulnerability	Description	Occurrence	Risk Level	CVE Num.
Read heap buffer overflow	forbidden memory access	denial of service attack	medium	requested
Write heap buffer overflow	Write memory heap buffer overflow	remote code execution attack	critical	requested

Table 5.3: Overview of found software bugs in suricata

The first bug is a forbidden memory access within the *IPv6-Decoder* more precisely in the *reassemble*-function. If we send two or more *IPv6* packets which are constructed in a way that the decoder has to reassemble all packets, the *DecodeIPV6ExtHdrs*-function in the file *decode-ipv6.c:347:25* tries to access memory regions that are not allocated. The *reassemble*-function doesn't check the length of the header while reassembling the IPv6 packets.

The second bug is more critical. We find a vulnerability in the AppLayerParser of the SSL protocol. The bug is located in the function *SSLv3ParseHandshakeType* in the file *app-layer-ssl.c:1476:13*. With a prepared ssl network packet we are able to overwrite a memory region by more than 772 bytes, which is enough to prepare an exploit for remote code execution.

5.2.3 Mosquitto MQTT-Broker Version 1.6.8

The **Eclipse Mosquitto** is an open source MQTT message broker that full fills the specification of the MQTT protocol version 5.0, 3.1.1 and 3.1. It is lightweight and is suitable for use on all devices from low power single board computers to full servers. The Mosquitto project also provides a C library for implementing MQTT clients, and the very popular mosquitto_pub and mosquitto_sub command line MQTT clients. [9]

The MQTT protocol is lightweight server-client message protocol which implements methods like *connect, disconnect, subscribe, unsubscribe* and many more. Overall MQTT has 14 specific methods which build on each other depending on the state. A standard connection setup consists of a *connect* packet, many *subscribe* and *publish* packets and a *disconnect* packet together. Usually MQTT runs over TCP/IP. The core application of this protocol is designed for IoT-devices like sensors and actuators.

Evaluation Target

The core task of the socket fuzz-target is to send a mutated data chain via a socket connection. We focus our experiments on real world instances and test the raw network interface of mosquitto. The following figure 5.7 describes the results and shows clearly the advantages of our approach.

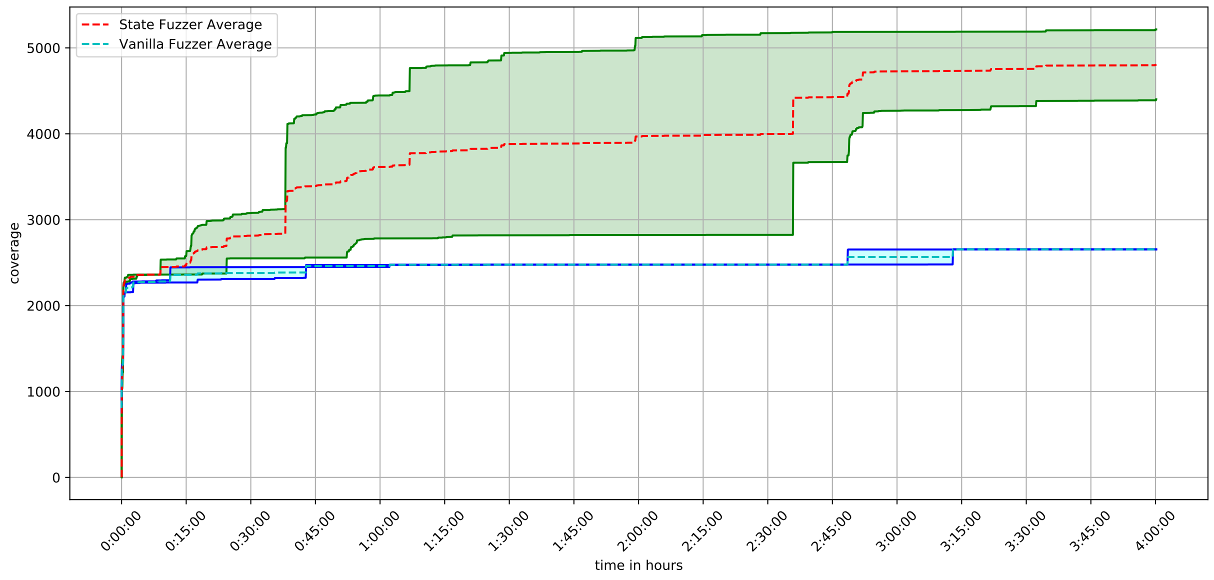
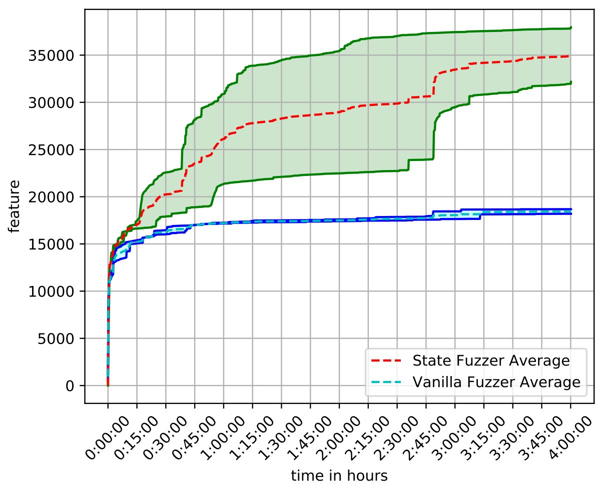
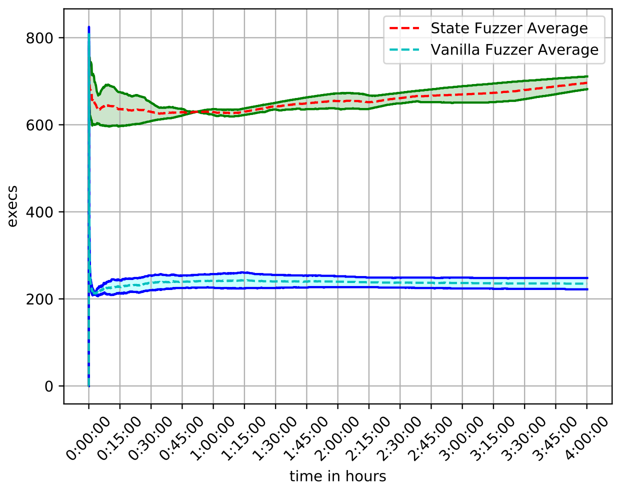


Figure 5.7: Mosquitto MQTT-Broker comparison of the code coverage between libFuzzer and the stateful fuzzer

This is an interesting experiment that show the huge advantage of the stateful fuzzer approach. Our fuzzer achieves in the best case more than 95% more code coverage and in the average case about 85%. The found features looks nearly the same with more than 90% in the average case. From the performance point of view we have attained more than twice as many executions per second. One reason for this is the connection establishment which works noticeably faster than



a) Comparison of found features



b) Comparison of the performance

Figure 5.8: Mosquitto MQTT-Broker comparison of found features and executions per seconds between libFuzzer and the stateful fuzzer

the error handling.

Findings

We found a bug in eclipse mosquitto. The following table 5.4 gives a short description of the bug. The core problem is a call of *memcpy()* in the file *lib/property_mosq.c* by passing a null-pointer

Type of vulnerability	**Description**	**Occurrence**	**Risk Level**	**CVE Num.**
Undef. behaviour	incorrect use of a function call	denial of service attack	low	requested

Table 5.4: Overview of found software bug in mosquitto

passed as argument 2, which is declared to never be null. The signature *memcpy()* is described as follows. The first parameter stands for the destination pointer, the second parameter is for the source address and the last parameter are the number of bytes to copy. If we prepare a modified network packet that triggers a *memcpy()* call but where the second parameter is a NULL pointer. In older *libc* versions this can be a problem and may cause the program to crash.

5.2.4 OpenSSL Version 3.0.0-dev

OpenSSL is a library for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library. It is used by many open source and closed source projects in the domain of secure network communication and cryptography applications.[27] We decided to test this project because it is a well-fuzzed open-source project. Furthermore, we want to be comparable to other fuzzers. The OpenSSL library is one of the projects of the *fuzzer-test-suite*, which is often used by scientists for fuzzing research.

Evaluation Target

The experiment tests the function *SSL_do_handshake()*. The fuzz-target is based on the target which was build to find the *Heartbleed Bug*. We adapt the code of the original fuzz-target so that it can handle the generated state chain. The following image 5.11 describes the performance of the stateful fuzzer library.

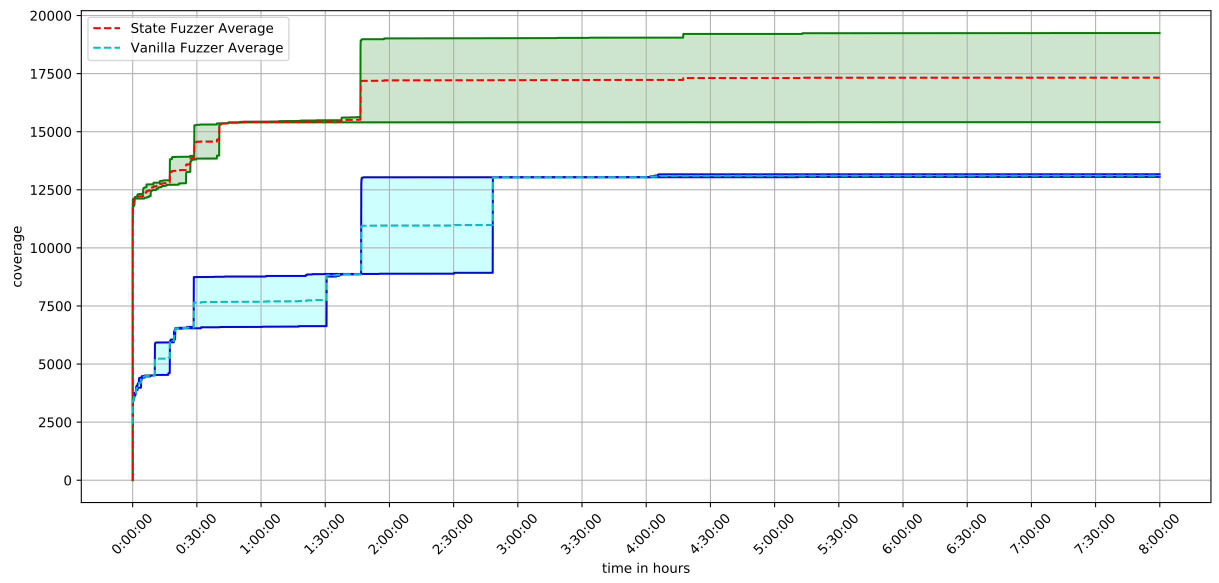
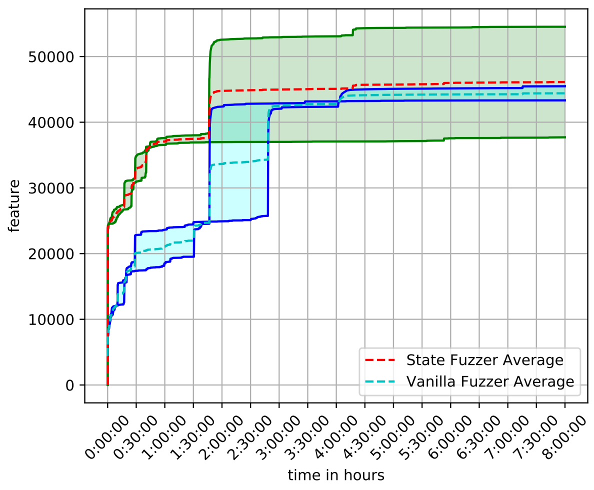
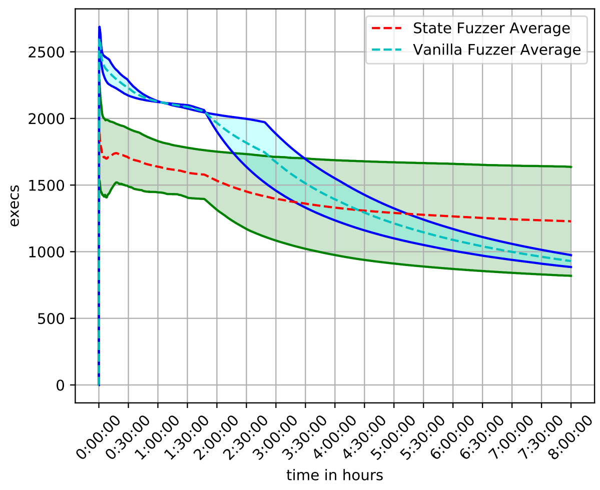


Figure 5.9: OpenSSL comparison of the code coverage between libFuzzer and the stateful fuzzer

The results demonstrate the effectiveness of the stateful fuzzer library. After only a few minutes we achieve more than twice as much code coverage than the fuzzer baseline (libFuzzer). In fact, we achieve in average 25% more code coverage after 8 hours of testing. The performance deviation of the stateful fuzzer is larger but at the end also faster than the usual fuzzer (see figure 5.12).



a) Comparison of found features



b) Comparison of the performance

Figure 5.10: OpenSSL comparison of found features and executions per seconds between libFuzzer and the stateful fuzzer

5.2.5 Lighttpd Webserver Version 1.4.54

The final experiment of our evaluation is the open source project *lighttpd*. Lighttpd is a lightweight http server and optimized for speed-critical environments. The first release of lighttpd took place in the year 2003 and was originally written by Jan Kneschke. Lighttpd is free and open-source software and is distributed under the BSD license.[20]

Evaluation Target

The evaluation test focuses on socked based communication just like the projects *libupnp* and *mosquitto*. We create fuzz-targets that are able to send the mutated data packets or the data chain via a socket connection.

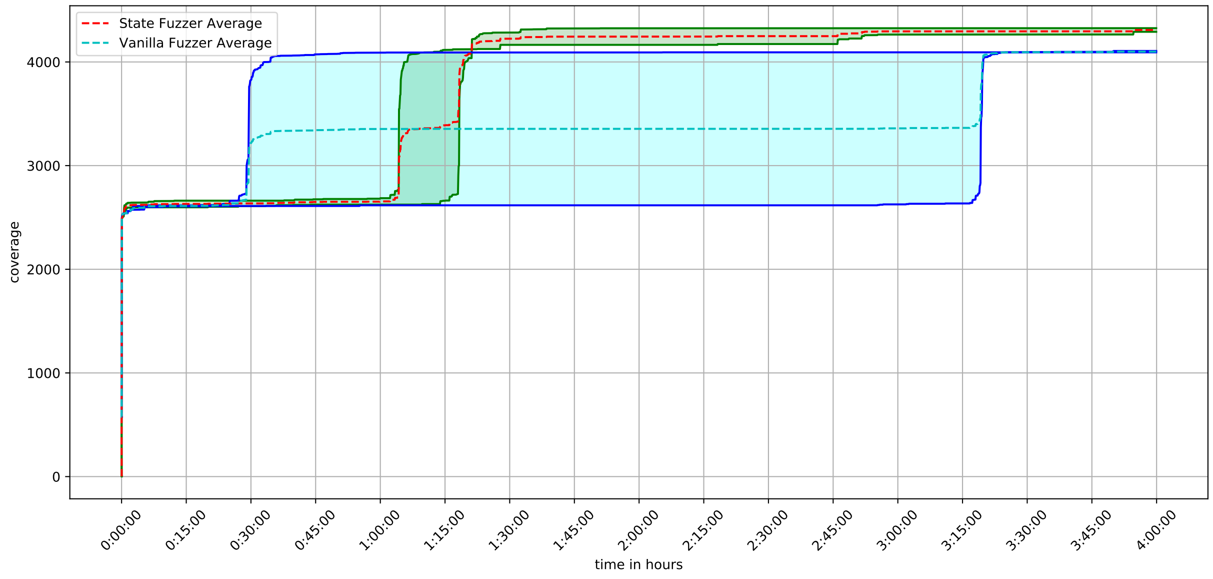
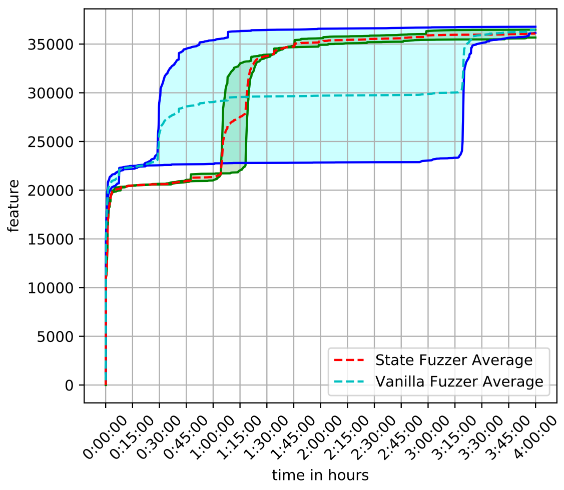
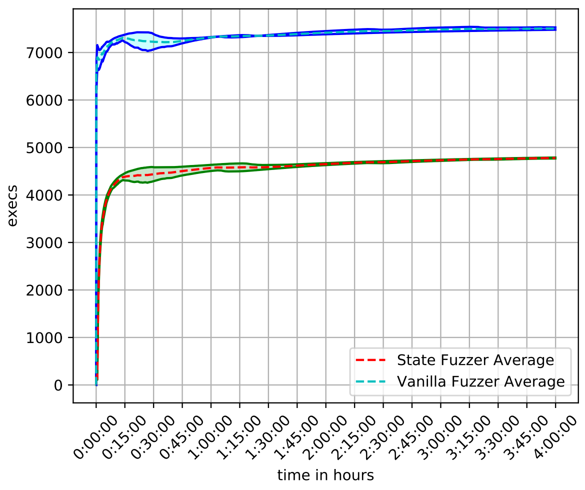


Figure 5.11: Lighttpd comparison of the code coverage between libFuzzer and the stateful fuzzer



a) Comparison of found features



b) Comparison of the performance

Figure 5.12: Lighttpd comparison of found features and executions per seconds between libFuzzer and the stateful fuzzer

In the results of the last experiment, we are approach is a little better than the baseline. We achieve in average 7% more code coverage after 4 hours of testing. The performance of the stateful fuzzer is nearly 50% is lower than the baseline, but we achieve at the end better code coverage. (see image 5.12).

5.3 Summary

After about 240 hours of testing we can summarize all data and improvements. All in all we improved the of code coverage by at least 6% to 15% depending on the fuzz-target. In average case we achieve 42% more code coverage over all tested projects. In some cases we improve the coverage by at most 89%.

Projects	**libFuzzer**	**Stateful Fuzzer**	**Improv-ment**
Libupnp	1514	1759	15%
Suricata (AppLayer)	4314	8153	89%
Suricata (TCP-Decoder)	585	617	6%
Mosquitto	2512	4670	85%
OpenSSL	13208	16980	28%
Lighttpd	4090	4310	7%
Sum	26223	36489	39%

Table 5.5: Overview of total number count of edges covered in average case

6 Conclusion

Fuzzing as a testing methodology is an auspicious technology that has been used to discover many important bugs related to security vulnerabilities. In the last few years many many researchers developed new algorithms to improve the results of fuzzing. Most researchers focused on enhancement of mutation algorithms such as FuzzerGym [8] or EnFuzz [7]. In this master thesis we improve not only the mutation algorithms but also introduce a new approach that generates packet chains depending on information of the coverage sanitizer and their grammar description. Based on the results of our benchmarks such as Google's fuzzer-test-suite and several real-world open source projects, we demonstrate that this approach of state dependent fuzzing outperforms the baseline libFuzzer up to 90% of code coverage. By using our interface and configuration of the state chain we are able to achieve substantial amount of code coverage specially for projects, which are condition dependent. However, this approach has also limitations when it comes to performance. But despite a lower execution speed we achieve at least the same or better coverage of the source code. In fact, we found new vulnerabilities in common software projects and demonstrate the effectiveness of our approach.

6.1 Future Work

Even though we are able to show that our approach works, there is a lot of space left for improvements. As we described, our solution generates the data fields within the data chain over the information of the coverage sanitizer. This information is distinct from each other. One improvement is to add new sanitizer the get a relationship between each data packet. Another possible improvement could be through new sanitizer which tracks syscalls like *read* or *write*. The information which are sent by *write* can be used by the mutator of libFuzzer or other fuzzing engines. With this information we could generate a data chain like a valid *SSL-Handshake*.

List of Figures

List of File

List of Tables

Bibliography

[1] Computer Networks CPS365 Fall 2016. Lab 4: Dns primer notes, 2016. *See http://www2.cs.duke.edu/courses/fall16/compsci356/DNS/DNS-primer.pdf*, accessed 2019-01-15.

[2] R. Sethi A. V. Aho, Monica S. Lam. *Compilers: Pearson New International Edition: Principles, Techniques, and Tools.* Addison Wesley, 2013.

[3] Dave Aitel. The advantages of block-based protocol analysis for security testing. In *The Advantages of Block-Based Protocol Analysis for Security Testing*, 111 E. 7. St. Suite 64, NY NY 10009, USA, 2002. Immunity, Inc.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 748–758. IEEE Press, 2019.

[5] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: Toward a stateful network protocol fuzzer. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 343–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Bryan So Barton P. Miller, Lars Fredriksen. *An Empirical Study of the Reliability of UNIX Utilities.* In An Empirical Study of the Reliability of UNIX Utilities. National Science Foundation, 1989.

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[8] William Drozd and Michael D. Wagner. Fuzzergym: A competitive framework for fuzzing and learning. In *FuzzerGym: A Competitive Framework for Fuzzing and Learning*, volume abs/1807.07490, 2018.

[9] Eclipse Foundation. Eclipse mosquitto$^{\mathrm{TM}}$, an open source mqtt broker, 2019. *See https://mosquitto.org/blog/*, accessed 2019-01-02.

[10] Open Information Security Foundation. Suricata open source ids / ips / nsm engine, 2010. *See https://suricata-ids.org/*, accessed 2019-01-02.

[11] Patrice Godefroid. Random testing for security: Blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, page 1, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Automated Whitebox Fuzz Testing*, 2006.

[13] Google. Security oriented fuzzer with powerful analysis options, 2019. *See https://github.com/google/honggfuzz*, accessed 2019-01-12.

[14] Protobuf Team Google. Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data., 2008. *See https://developers.google.com/protocol-buffers*, accessed 2019-01-02.

[15] David A. Patterson John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[16] Matt Morehouse K. Serebryany, Vitaly Buka. libprotobuf-mutator, 2017. *See https://github.com/google/libprotobuf-mutator/blob/master/src/mutator.cc*, accessed 2019-01-02.

[17] Rauli Kaksonen, Marko Laakso, and Ari Takanen. *Software Security Assessment through Specification Mutations and Fault Injection*, pages 173–183. Springer US, Boston, MA, 2001.

[18] Eoin Keary. 2019 vulnerability statistics report. In *2019 Vulnerability Statistics Report*. edgescan, 2019.

[19] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Jan Kneschke. Lighttpd fly light, 2003. *See https://www.lighttpd.net/*, accessed 2019-01-02.

[21] Google LLVM Team. Sanitizercoverage, 2011. *See https://clang.llvm.org/docs/SanitizerCoverage.html*, accessed 2019-01-02.

[22] Google LLVM Team. libfuzzer – a library for coverage-guided fuzz testing, 2016. *See https://llvm.org/docs/LibFuzzer.html*, accessed 2019-01-02.

[23] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[24] John Grosen Nick Stephens. Driller: Augmenting fuzzing through selective symbolic execution. In *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*, UC Santa Barbara, 2016. Seclabs.

[25] Michel Pfeiffer et al. Portable sdk for upnp devices, 2019. *See https://pupnp.sourceforge.io/*, accessed 2019-01-02.

[26] Srđan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslic. Performance evaluation of using protocol buffers in the internet of things communication. In *Performance evaluation of using Protocol Buffers in the Internet of Things communication*, pages 261–265, 10 2016.

[27] The OpenSSL Project. Openssl cryptography and ssl/tls toolkit, 1998. *See https://www.openssl.org/*, accessed 2019-01-02.

[28] K. Serebryany. Sanitize, fuzz, and harden your c++ code, 2011. *See https://www.usenix.org/sites/default/files/conference/protected-files/enigma_slides_serebryany.pdf*, accessed 2019-01-02.

[29] K. Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, Nov 2016.

[30] Kartik Trivedi. Foundstone wsdigger 1.0. In *Foundstone WSDigger 1.0*. McAFEE, 2005.

[31] Michał Zalewski. american fuzzy lop (2.52b), Juni 2013. *See http://lcamtuf.coredump.cx/afl/technical_details.txt*, accessed 2020-01-03.