Karlsruhe Institute of Technology

Fraunhofer IOSB

Design and Evaluation of Methods for Efficient Fuzzing of Stateful Software

Masterthesis

KIT – Karlsruher Institut für Technologie
Fraunhofer IOSB – Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung

Mark Giraud

31 December 2020

Responsible supervisor: Prof. Dr.-Ing. habil. Jürgen Beyerer
Supervising employee: Dr.-Ing. Christian Haas
MSc Anne Borcherding

Interaktive Echtzeitsysteme IES

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 31 December 2020

(Mark Giraud)

Abstract

Due to increasing connectivity, server software as well as embedded controllers are becoming ever more prevalent. As such, securing these components against attacks by finding bugs early in development is becoming more important. In recent years, fuzz testing has become increasingly popular as an automated testing method for this purpose. Most fuzzing approaches however only consider targets that process a single input and then exit, limiting their suitability for highly stateful software like servers or embedded controllers.

This thesis analyses the most recent developments in stateful fuzzing and identifies automatic state identification as a possible improvement to the existing AFLNet, which currently requires manual specification. A novel approach for automatic state identification (SNAPP) is proposed, that uses the coverage information already provided by coverage guided fuzzers. The approach is implemented and evaluated on the three stateful targets open62541, lightftp, and live555. The obtained results show that SNAPP performs at least as well and in certain cases better than AFLNet, without needing a manually constructed input specification in order to extract states. The results also suggest that initial seed choice plays a key role in stateful fuzzer performance.

Zusammenfassung

Aufgrund der zunehmenden Vernetzung von Server-Software und Embedded-Controllern gewinnt das frühzeitige Testen dieser Systeme auf Sicherheitslücken bereits während der Entwicklung an Relevanz. In den letzten Jahren hat Fuzz-Testing zu diesem Zweck zunehmend an Bedeutung gewonnen. Die meisten Fuzzing-Verfahren wurden jedoch entwickelt, um Systeme zu testen, die eine einzelne Eingabe entgegennehmen und sich dann beenden. Dadurch lassen sich diese Fuzzer in der Regel nur bedingt auf zustandsbehaftete Server-Software oder Embedded-Controller anwenden.

Diese Arbeit analysiert die neuesten Entwicklungen im Bereich Fuzzing zustandsbehafteter Software. Die automatische Zustandserkennung wird als mögliche Verbesserung zu dem bereits existierenden Fuzzer AFLNet identifiziert, welcher aktuell eine manuelle Spezifikation benötigt, um Zustände zu extrahieren. Diese Arbeit konzipiert einen neuen Ansatz zur automatischen Extraktion von Zuständen (SNAPP), der die Coverage-Information verwendet, die Coverage-Guided-Fuzzern bereits zur Verfügung steht. Der Ansatz wird implementiert und auf den drei zustandsbehafteten Testzielen open62541, lightftp, und live555 evaluiert. Die durch die Evaluation erzielten Ergebnisse zeigen, dass SNAPP in den meisten Fällen mindestens genauso gut wie AFLNet und in manchen Fällen sogar besser abschneidet. Dabei benötigt SNAPP im Gegensatz zu AFLNet keine manuelle Spezifikation um Zustände zu extrahieren. Die Ergebnisse deuten auch darauf hin, dass die initiale Wahl des Corpus ausschlaggebend für die Performance der zustandsbehafteten Fuzzer ist.

Notation

$\text{beh}_{\mathcal{P}}(s,m)$	The behavior of the program $\mathcal{P}$ for message m after processing sequence s
$\langle m_1, \ldots, m_n \rangle$	Sequence of messages
$x^{(i)}$	Element i of tuple x. Indexing starts at 1
$\mathfrak{S}$	Set of states
$\mathfrak{s}$	A state in $\mathfrak{S}$
$\mathcal{S}$	Set of sequences in a state $\mathfrak{s}$
$\mathcal{B}$	Set of observed behaviors in a state $\mathfrak{s}$
$\mathfrak{b}$	An observed behavior $\mathfrak{b} \in \mathcal{B}$

Contents

1 Introduction

In an increasingly digitalized and interconnected world, the relevance of secure IT systems is becoming more important than ever. Major security relevant bugs like Heartbleed [HB14; CVE13], Stuxnet [FMC11], or Shellshock [Sel14; CVE14] show that even simple undiscovered bugs can have a large impact. Most recently, the network management tool Orion, manufactured by SolarWinds, was breached [Goo20a]. The breach exploited bugs in order to gain access to SolarWinds' build system, making it possible to execute a supply chain attack. In this case, the attackers were able to inject a backdoor into software updates distributed by SolarWinds to a plethora of companies and US government agencies [Goo20b].

In many cases, the causes for these breaches are bugs in parts of the software's code. Fuzz testing, i.e. automatically supplying inputs generated by a tool (fuzzer) to a system under test (SUT) and monitoring the SUT for crashes, has been shown to be able to find bugs abused for breaches like the aforementioned Heartbleed bug [Whe17]. As such, the Heartbleed bug could have been prevented by fuzzing the openssl library. Since the inception of fuzzing more than 30 years ago [Tak+18], even the most basic fuzzing techniques have not diminished in relevance as shown by Miller et al. in a recent study [MZH20].

Recent fuzzing research, however, mostly considers software that processes a single input and then terminates. Although these approaches are applicable in a limited way to highly stateful software like servers or embedded software, Pham et al. [PBR20b] recently showed that by considering the stateful nature of the SUT, fuzzing performance can be significantly improved. Their approach AFLNet uses a simple protocol specification in order to split inputs into individual messages and extract response codes in order to build a state machine that is used to further guide the fuzzer. Except for blackbox approaches, stateful software fuzzing has not received much attention until the recent publication of AFLNet. This thesis will investigate methods for efficient fuzzing of stateful software on the basis of AFLNet as current state of the art.

This thesis was created in cooperation with and sponsored by Robert Bosch GmbH.

1.1 Objective

Fuzzing more types of software and providing better usability, e.g. automating more steps in the fuzzing process, is identified by Böhme et al. [BCR20] as one of many current challenges. They consider especially fuzzing stateful software as open research. As previously discussed, stateful blackbox fuzzing has already been looked into [Dou+12; RP15; Fit+20; AGP19; Ma+16; Gas+15], whereas stateful greybox fuzzing has only recently been investigated by Pham et al. [PBR20b]. Due to more insight into the targets, greybox fuzzing in most cases is able to perform better than blackbox fuzzing, and as such we will analyze the fuzzer AFLNet by Pham et al. [PBR20b] to identify possible improvements. This thesis mainly aims to improve the adaptation of fuzzing for stateful software by reducing the required amount of manual work when adapting to new targets. This thesis will propose and implement a possible solution that reduces the manual work required. This is achieved by using the already provided coverage feedback of AFLNet, instead of responses that need to be parsed, and therefore omitting the need for a parser. To compare the implemented solution SNAPP against the state of the art, we will evaluate SNAPP against AFLNet on the three SUTs open62541, lightftp, and live555. In addition, we will analyze the impact of seed selection for stateful fuzzing and look at possible performance increases by using snapshotting techniques.

1.2 Outline

The thesis is structured into seven parts. In Chapter 2, the concepts used in this thesis are briefly introduced. Afterwards, challenges in stateful fuzzing and the current state of the art are discussed in Chapter 3, and we pose four research questions. Using this analysis as basis, Chapter 4 then introduces a novel automatic state identification algorithm, and the framework required for the algorithm to work. The implementation of the proposed techniques is then discussed in Chapter 5. Next, the evaluation of the implemented fuzzer is presented in Chapter 6 by first discussing evaluation choices and then providing the achieved results. In Chapter 7, each of the four research questions will be answered on the basis of the achieved results. In addition, some additional insights and limitations of the current approach will be discussed, and the thesis is put into context by discussing related work. The chapter is concluded by directions for future work and an overall summary of the results. Finally, Chapter 8 concludes this thesis by summarizing the most important aspects.

2 Background

This chapter contains some background on fuzzing, and a few definitions to establish a framework in order describe the concepts used in this thesis. We first briefly discuss different state types and differentiate their meaning in Section 2.1. Next, we introduce a small example program that is used throughout this thesis in Section 2.3. In Section 2.4, we then explain fuzzing and differentiate between different fuzzer types. Afterwards, we discuss coverage guided fuzzing in detail and introduce definitions for coverage and related concepts in Section 2.5. Finally, we differentiate different types of SUTs in Section 2.6.

2.1 State Concepts

In the context of fuzzing, a state can refer to multiple concepts. In order to avoid confusion later on, we differentiate between these concepts in this section. Salls et al. [Sal+20] analyse current approaches and generalize the state concept by differentiating between two state concepts. First, they introduce a *concrete state* as "the snapshot of all processor registers, the program's memory, file system operations, or anything else that effects the operation of the program" [Sal+20]. This means that each time the SUT executes a single operation, the SUT's *concrete state* changes. An input to the SUT thus produces a sequence of *concrete states*, called a *concrete state trace*. The set of all possible *concrete state traces* is called the *concrete state space*. Salls et al. then introduce an abstraction function used by fuzzers in order to map the *concrete state space* to an *abstract state space*. This mapping could for example be the coverage obtained after executing a certain input.

In the context of protocols, states often are a high level abstraction of the *concrete state space*. For our purposes, we will use the term state to refer to a set of *concrete state traces* that exhibit "similar" behavior of the SUT. The meaning of "similar" will become clear when the state identification algorithm is introduced in Chapter 4.

2.2 OPC UA

OPC UA is a platform independent communication standard for machine to machine communication over various types of networks [Fou17a]. It supports secure communication with different security policies and also provides measures to assure the identity of OPC UA applications [Fou17a]. OPC UA uses a server client model, where servers provide services that a client may use. Similar services are grouped together into service sets. All service sets are listed in Table 2.2. A server instance however is not required to provide all of these services.

Service Set	Description
Discovery	Discover Endpoints provided by a Server
SecureChannel	Establish a secure communication channel
Session	Provides user authentication and manages sessions
NodeManagement	Manage nodes in a server's address space
View	Browse a server's address space and subsets of it
Query	Query a set of data in the server's address space
Attribute	Read and write attributes of nodes
Method	Call remote procedures in the server's address space
MonitoredItem	Manage monitored items used to monitor attributes
Subscription	Manage subscriptions to which monitored items can be attached

Table 2.2: Service sets of the OPC UA standard [Fou17a].

We will briefly describe the services in the following paragraphs. For a more detailed description of the services the standard should be consulted [Fou17b].

All services listed in Table 2.2, except for the SecureChannel service, require an established SecureChannel. Most of the services also require the user to be authenticated, meaning that a session has to be established. Before any of the services can be used by a client, the client needs to establish a connection via TCP, HTTPS, or Websockets. Afterwards, the client needs to establish a secure communication channel by using the SecureChannel service set. For services that require user authentication, the client then needs to establish a session by authenticating itself with credentials.

The Discovery service set is used by clients to get information about a server's endpoints. An endpoint is a possible connection type, e.g. TCP, and a corresponding security configuration, e.g. unencrypted. The Discovery service set also contains services to find servers by using a commonly known discovery server, to which other servers can register themselves by using services part of this service set.

OPC UA servers have a so called address space that contains nodes. Clients can create, modify, and delete nodes by using the services in the NodeManagement service set. Nodes also have attributes that clients can read, or write by using the services in the Attribute service set. Nodes might also have methods associated with them that a client can call by using the Method service set. The View and Query service sets are used by clients in order to browse a server's address space and to get subsets of the data in the address space. Finally, clients may use the MonitoredItem and Subscription service sets in conjunction to monitor attributes of nodes in the server's address space. In order to do so, a client first creates a subscription by using the services provided by the Subscription service set. Then, the client creates monitored items for attributes that it wants to monitor by using the services in the MonitoredItem service set. These monitored items are then attached to the subscription. The server sends updates about the monitored items in the subscription to the client with a periodic interval that can be configured by the client.

The OPC UA standard also defines a publish/subscribe communication model. We only consider the previously described services that use the client/server communication model in this thesis.

In this thesis we will use the open source implementation open62541 of the OPC UA protocol [Iat+20]. The implementation supports most of the services specified by the OPC UA standard. Most importantly, it supports the highly stateful NodeManagement, Attribute, MonitoredItem, and Subscription services, which are relevant for the evaluation in Chapter 6.

2.3 Code Example

In this section, we will look at a small example program that we will use throughout this thesis in order to explain the concepts used. For ease of understanding we will use a python like pseudo code instead of `c` or `c++`.

We will look at a simplified OPC UA protocol with three message types: OPN, MSG and ERR. The OPN message is responsible for establishing a connection. It has to be sent first, in order for any MSG messages to be processed in a meaningful way by the SUT. A MSG message triggers some kind of processing by the SUT in the `do_msg` function. In the real OPC UA protocol this is subdivided into many different message types. For simplicity, we will only concern ourselves with the `CreateNode` and `DeleteNode` messages, and the connection phase. For the `CreateNode` MSG and `DeleteNode` MSG there will be no pseudocode. It is only relevant that a `CreateNode` message will create a node with a supplied id. The `DeleteNode` message will remove the node with the supplied id if it exists. In case of an unrecognized message, our example program will simply send an ERR message.

```
1  connected = False
2  while True:
3      msg = recv(timeout=0.05)
4      if msg.typ == "OPN":
5          response = do_opn(msg)
6          connected = True
7      elif msg.typ == "MSG" and connected:
8          response = do_msg(msg)
9      elif msg.typ == "CLO" and connected:
10         response = do_msg(msg)
11         connected = False
12     elif msg is not None:
13         response = "ERR"
14     if response is not None:
15         send_response(response)
16     do_background_processing()
```

Listing 2.1: Example Program based on OPC UA, modelling a simple server.

The example code modelling this simplified OPC UA protocol can be seen in Listing 2.1. Usually a server will have to handle new TCP connections and accept them. For simplicity, we only model the code for an already accepted TCP connection. The `recv` function in Line 3 will block for `timeout` seconds and try to receive a complete message. If a message is received, it is returned. Otherwise, if the timeout is reached, `recv` will return `None`. The `send_response` function in Line 15 simply sends the response to the connected client over the TCP connection.

The global variable `connected` (Line 1) determines if a connection is open. It is initially set to `False`. An OPN message will set the `connected` variable to `True`. All following messages of type MSG and CLO can then be processed. A CLO message will reset the `connected` variable to `False`. In case a message has an incorrect type, or was sent in an incorrect connection state, the program sends an ERR message (Line 12).

In each main loop iteration the program also calls a function `do_background_processing` that performs some arbitrary processing in each main loop iteration. For our purposes it is only important that such processing might occur in each iteration of the mainloop.

The state machine depicted in Figure 2.1 shows the two states this simple protocol can assume. The transitions depict the received message, and the output that is produced. Invalid messages will result in a self loop, and an ERR message as output, which is omitted in order to not clutter the figure.

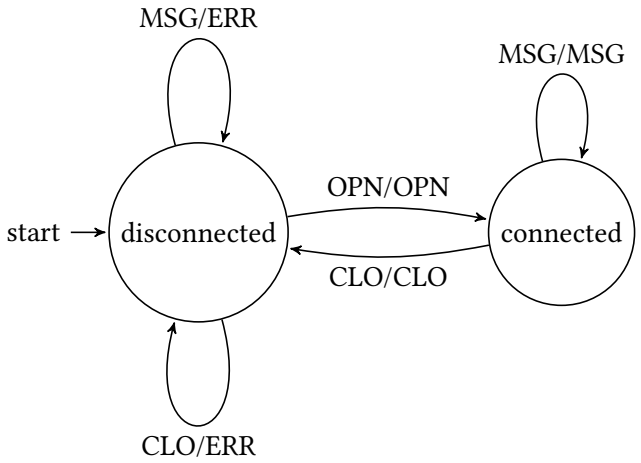


Figure 2.1: The protocol state machine of the simplified OPC UA protocol.

It should be noted that this state machine models the protocol state behavior, i.e. establishment of a connection. The program has further states when creating and deleting nodes. For example, if a node with id 1 is created, the program will reflect this in an internal state, such that it can be deleted at a later point in time. As mentioned in Section 2.1 we will later propose an approach for an abstraction that does not only identify the protocol states, but is also able to identify stateful behavior like the one mentioned previously.

2.4 Fuzzers

Fuzzing is the process of testing an SUT by supplying it with automatically generated inputs, and observing the SUT's behavior on the generated inputs, in oder to discover bugs. We will adapt the description of a general fuzzing algorithm from Manès et al. [Man+19]. The general fuzzing algorithm is depicted in Algorithm 1. A fuzzer usually takes an initial set of fuzz configurations $\mathbb{C}$, and a maximum time t_{limit} the fuzzer should run for as input. A fuzz configuration consists of the parameters for the fuzz algorithm. It can for example be an input that is modified by the fuzzer, or an input format specification that is used to generate inputs. If an input is contained in a fuzz configuration, this input is usually called a seed. The set of fuzz configurations is often referred to as corpus, if it primarily consists of seed inputs. The maximum time t_{limit} may be infinite, if for example the fuzzer should run until interrupted by a user. The output of a fuzzer is usually a set of bugs $\mathbb{B}$ found during fuzzing.

Algorithm 1: A generic fuzzing algorithm. (adapted from [Man+19])

Input : $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$

```
1 B ← ∅;
2 C ← Preprocess(C);
3 while t_elapsed < t_limit ∧ Continue(C) do
4 |   conf ← Schedule(C, t_elapsed, t_limit);
5 |   tcs ← InputGen(conf);
6 |   B′, execinfos ← InputEval(conf, tcs, O_bug);
7 |   C ← ConfUpdate(C, conf, execinfos);
8 |   B ← B ∪ B′;
9 return B
```

Preprocess($\mathbb{C}$) $\rightarrow \mathbb{C}$

The Preprocess function takes a user supplied set of initial fuzz configurations and returns a possibly modified set. Modifications can include instrumenting the SUT or precalculating metrics on initial seed inputs.

Schedule($\mathbb{C}$,$t_{elapsed}$,t_{limit}) $\rightarrow$ `conf`

The Schedule function takes a set of fuzz configurations, and the elapsed and maximum time as parameters. It selects a fuzz configuration to be used for the current fuzz iteration.

InputGen(`conf`) $\rightarrow$ `tcs`

The InputGen function takes the selected fuzz configuration and generates new test cases using this configuration. If, for example, the fuzz configuration consists of an input specification, this specification is used by the function to generate new inputs as test cases. Another common scenario is the use of seed inputs as fuzz configurations. In this case, the seed inputs are modified by a mutation algorithm in order to generate new test cases.

InputEval(`conf`,`tcs`,O_{bug}) $\rightarrow$ $\mathbb{B}'$,`execinfos`

The InputEval function takes the selected fuzz configuration, and the test cases generated from it as input. In addition, the function is supplied a bug oracle. The function executes the SUT on all test cases and uses the bug oracle to determine whether the execution of a test case produces a bug. The output of the function is a set of found bugs $\mathbb{B}'$, and a set of `execinfos` that contains information about each test case execution.

ConfUpdate($\mathbb{C}$,`conf`,`execinfos`) $\rightarrow \mathbb{C}$

The ConfUpdate function takes the set of fuzz configurations $\mathbb{C}$, the fuzz configuration

`conf` used in the current fuzzing iteration, and the `execinfos` produced by running the SUT on the generated test cases as input. The function modifies the set of fuzz configurations depending on the information acquired when executing the test cases. Some fuzzers for example use information about execution times to trim the set of fuzz configurations. It is also possible that the function returns the set of fuzz configurations unchanged.

Continue($\mathbb{C}$) $\rightarrow$ {*True*,*False*}

The Continue function takes the set of fuzz configurations $\mathbb{C}$ and determines whether to continue fuzzing. Criteria can for example be that a certain coverage threshold was reached, or that all possible paths were exhausted.

Fuzzers are usually categorized into whitebox, blackbox, and greybox fuzzers. We will now briefly discuss the differences of these categories.

2.4.1 Blackbox Fuzzing

The term blackbox fuzzing refers to fuzzing techniques that have no knowledge about the internals of the SUT. The only way for a fuzzer to interact with the SUT is via the SUT's input and output interfaces as depicted in Figure 2.2. The depicted fuzzer uses the generic fuzzing algorithm described in Algorithm 1. During the InputEval step, a blackbox fuzzer can only send and receive data to and from the SUT's I/O interfaces.

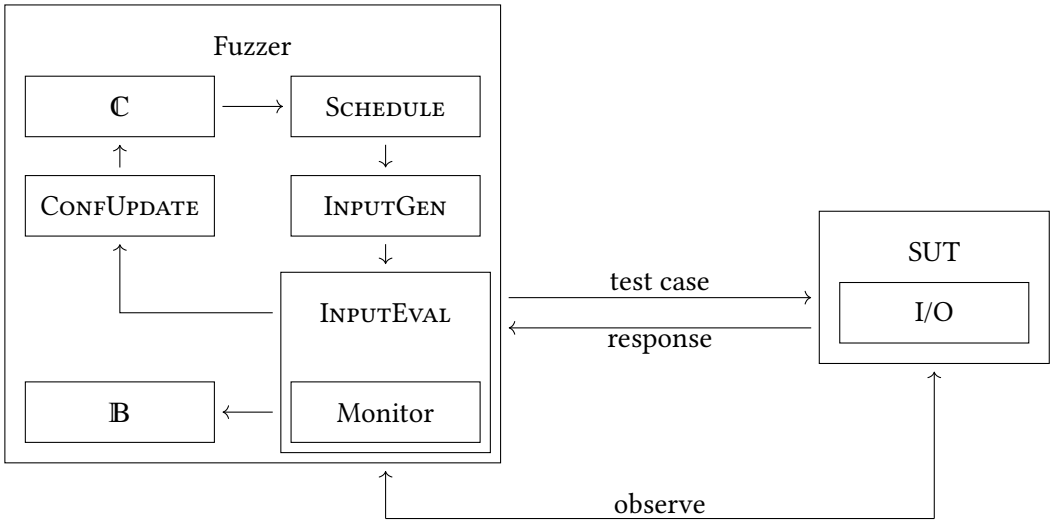


Figure 2.2: Blackbox fuzzing using the generic fuzzing algorithm. (Algorithm 1)

Bug detection also has to be done by observing the I/O behavior of the SUT. For example, a monitor might be used, that sends an input to the SUT and observes the response, checking its validity against a reference. If the response is not received, or if it is invalid, the SUT can be considered to have encountered a bug. Also, it might not be possible to reset the SUT after each input, making it difficult to determine if a single input caused the SUT to fail, or if a combination of inputs was responsible for the failure.

Although blackbox fuzzing has its limitations, it still has its place. It can be used to fuzz remote SUTs that the user has no other access to, for example if the SUT is delivered as an already assembled hardware device. Furthermore, blackbox techniques are also useful, when SUTs need to run on embedded hardware and limited resources make instrumentation difficult. The work by Manès et al. [Man+19] gives an overview of most fuzzers developed up until the paper's publication, including blackbox approaches. We will also cover some blackbox approaches related to this thesis in Section 7.4.

2.4.2 Whitebox Fuzzing

In contrast to blackbox fuzzing, whitebox fuzzing has access to all internals of the SUT as depicted in Figure 2.3. This includes for example the SUT's source code. This way whitebox fuzzers are able to perform systematic exploration of the SUT's *concrete state space* (see Section 2.1). One approach is to use symbolic execution in order to perform such an exploration of the *concrete state space* by assuming symbolic values instead of concrete inputs and using a satisfiability modulo theories (SMT) solver to find inputs that exercise a certain path. However, since pure symbolic execution results in an explosion of possible *concrete state traces*, most approaches combine symbolic execution with concrete execution of the SUT. Whitebox fuzzing in most cases comes with a much higher overhead than blackbox fuzzing, since whitebox fuzzing often makes use of SMT solvers and other resource intensive techniques [Man+19]. We discuss whitebox approaches related to this thesis in Section 7.4.

2.4.3 Greybox Fuzzing

Greybox fuzzing approaches sacrifice some of the information available to whitebox approaches, in order to increase fuzzing performance. These approaches often perform lightweight static analysis on the SUTs, or inject lightweight instrumentation to the SUTs in order to extract coverage information or other easy to extract statistics. These analyses or instrumentation techniques provide an approximation of the SUT's behavior that is used by the fuzzing algorithm to generate new test cases more quickly than whitebox approaches would be able to. Notable

examples for greybox fuzzers are AFL [Zal+20] and libFuzzer [LLV20]. Both of these approaches perform coverage guided fuzzing, which we will describe in more detail in the following section.

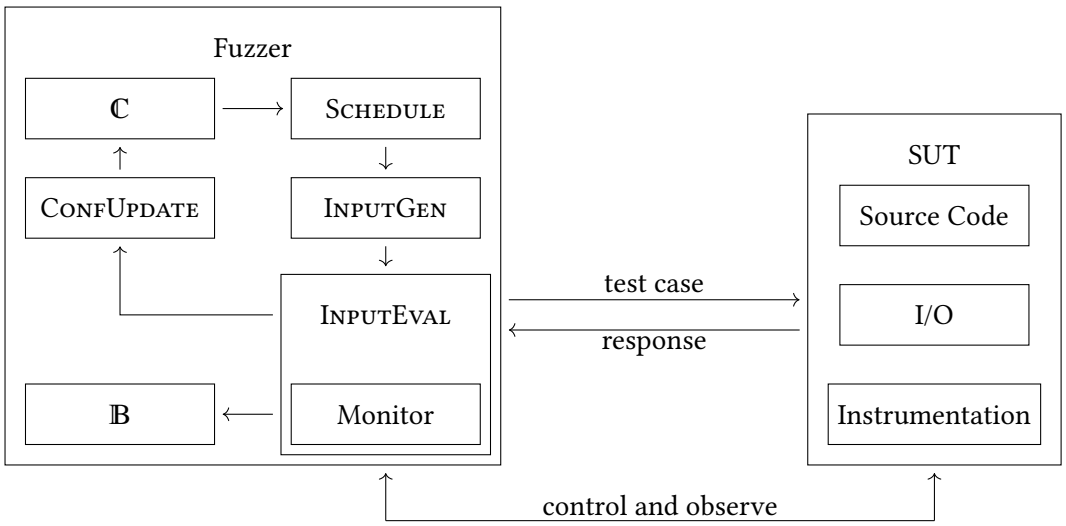


Figure 2.3: Whitebox fuzzing using the generic fuzzing algorithm. (Algorithm 1)

2.5 Coverage Guided Fuzzing

As the name suggests, coverage guided fuzzing uses coverage feedback of the SUT in order to guide the fuzzer. Coverage guided fuzzers perform instrumentation in the Preprocess function. The instrumentation gathers information about the executed code regions during runtime of the SUT when processing an input. We will use AFL as a representative example for most coverage guided fuzzers, since other coverage guided approaches are mostly derived from either AFL or libFuzzer and libFuzzer is similar to AFL [Man+19].

AFL starts with a set of fuzz configurations ℂ that consists of initial seeds supplied by the user, and the SUT. The Preprocess function then instruments the SUT such that it gathers information about which code regions of the SUT were executed. AFL then chooses a seed to process by calling the Schedule function in the main fuzzing loop. The chosen seed is then mutated in the InputGen function by a set of mutators. These mutators perform operations like bit flips, inserting words from dictionaries, or entirely random mutations. Afterwards, the input generated by mutating the original seed is executed on the SUT. AFL then observes if the SUT crashes, and gathers the coverage feedback. Finally, if a crash was observed, the crashing input is added to the set of bugs 𝔹. Otherwise, the ConfUpdate function checks if any code

regions that were previously not covered are now covered. If this is the case, the input is added to the set of fuzz configurations $\mathbb{C}$ as a seed input. Otherwise, AFL simply discards the input.

Since AFL evolves a set of seeds (the corpus) during its execution, it is also called an evolutionary fuzzer. In contrast to AFL and other coverage guided fuzzers, some fuzzers are classified as generative fuzzers, since they use an input specification in order to generate messages according to the specification. However, the approaches are not exclusive and there exist approaches, combining evolutionary and generative approaches [WLR20]. In this thesis we will extend AFLNet, a greybox fuzzer based on AFL, which also uses a combined approach of generative and evolutionary fuzzing [PBR20b]. In order to understand the concepts used later on in Chapter 4, we will formally define coverage in the following sections.

2.5.1 Control Flow Graphs and Basic Blocks

In order to describe the behavior of a program it is useful to model it as a control flow graph (CFG). The code of a program can be split into so called basic blocks. We will define basic blocks and CFGs similar to Torczon and Cooper [TC07].

Definition 2.1 (Basic Block) *A basic block is a maximal length sequence of branch free code. It begins with a labelled operation and ends with a branch, jump, or predicated operation.*

Consider the example program in Listing 2.1. In order to extract its basic blocks, we first transform the code into an assembly-like pseudo code as seen in Listing 2.2 The basic blocks are marked in alternating colors as seen in Listing 2.2. Each of the blocks ends with an explicit jump instruction, or implicitly, because the next line is a label and thus marks the start of a basic block. The `jne` instruction for example would produce two edges: One edge from the first basic block to the second basic block, corresponding to the path when the jump is not taken, and another edge from the first basic block to the third basic block, corresponding to the path when the jump is taken. This concept of basic blocks leads us to the definition of the CFG.

Definition 2.2 (Control Flow Graph) *A CFG is a directed graph $G = (B,E)$. Each vertex $b \in B$ corresponds to a basic block of the program. Each edge $e = (b_i, b_j) \in E$ corresponds to a possible transfer of control from basic block b_i to basic block b_j.*

The example program in assembly form in Listing 2.2 results in the CFG depicted in Figure 2.4. Each node label corresponds to the line number of the first line of the corresponding basic block.

```
    connected = False
loop:
    msg = recv(timeout=0.05)
    cmp msg.typ == "OPN"
    jne notOPN
    response = do_opn(msg)
    connected = True
    jmp endif
notOPN:
    cmp msg.typ == "MSG" and connected
    jne notMSG:
    response = do_msg(msg)
    jmp endif
notMSG:
    cmp msg.typ == "CLO" and connected
    jne notCLO
    response = do_msg(msg)
    connected = False
    jmp endif
notCLO:
    cmp msg == None
    je endif
    response = "ERR"
endif:
    cmp response == None
    je endif2
    send_response(response)
endif2:
    do_background_processing()
    jmp loop
```

Listing 2.2: Example Program based on OPC UA, modelling a simple server.

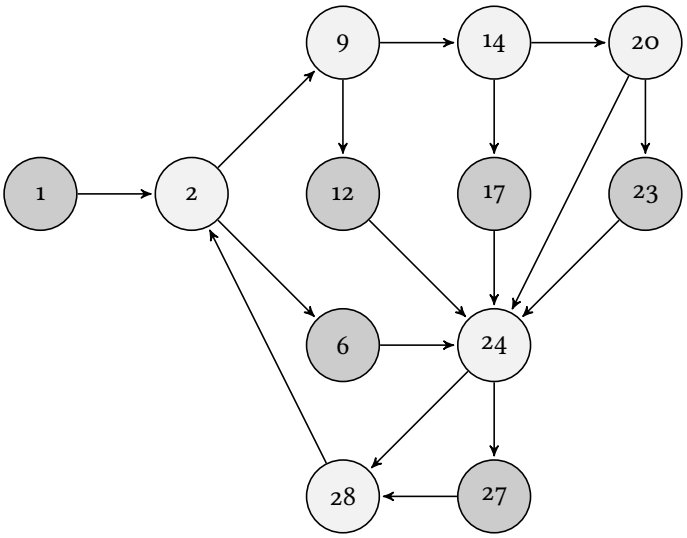


Figure 2.4: The CFG representing Listing 2.2. Node labels represent the first line number of the corresponding basic block.

A concrete execution of a program then corresponds to a potentially infinite path through the CFG $p = (e_1, e_2, \dots)$ where $\forall e_i = (b_i^1, b_i^2), e_j = (b_j^1, b_j^2) \in p : j = i + 1 \Rightarrow b_i^2 = b_j^1$. For example, if the example program never receives any input, it will loop forever following the path $p = ((1,2),(2,9),(9,14),(14,20),(20,24),(24,28),(28,2),(2,9),\dots)$. We will now use the concept of a path to define path basic block coverage as the amount of times a basic block is hit for a path taken when executing the program.

Definition 2.3 (Path Basic Block Coverage) *Let $P \subseteq E^*$ be the (infinite) set of possible paths a program can take and $G = (B, E)$ the CFG that represents the program $\mathcal{P}$. Let $C = \mathbb{N}^{|B|}$ be the set of possible observed basic block coverage profiles. Then the basic block coverage of a path $p \in P$ is defined as*

$$\begin{aligned} \text{pbcov}_{\mathcal{P}} &: P \to C \\ p = (e_1, e_2, \dots) &\mapsto \left(c_i \mid c_i = \left|\left\{e \in p \mid e = (b^1, b^2) \wedge b^1 = b_i\right\}\right|\right) \end{aligned}$$

In addition to basic block coverage, edge coverage is also interesting, since it provides more fine-grained information. The definition is very similar to that of basic block coverage. Instead of counting the amount each basic block was hit on the path p, we count the number of times an edge was traversed.

Definition 2.4 (Path Edge Coverage) *Let $P \subseteq E^*$ be the (infinite) set of possible paths a program can take and $G = (B, E)$ the CFG that represents the program $\mathcal{P}$. Let $C = \mathbb{N}^{|E|}$ be the set of possible observed edge coverage profiles. Then the edge coverage of a path $p \in P$ is defined as*

$$\begin{aligned} \text{pecov}_{\mathcal{P}} &: P \to C \\ p = (e_1, e_2, \dots) &\mapsto (c_i \mid c_i = |\{e \in p \mid e = e_i\}|) \end{aligned}$$

Since in most cases we will be interested in the coverage a certain input to the program produces, we will further define the coverage functions to take an input instead of an execution path. First we will define a function to map an input to the execution path the program will take, given this input.

Definition 2.5 (cfgpath) *Let $\mathcal{P}$ be a deterministic program that is represented by the CFG $G = (B, E)$. Let $\iota \in I = \{0,1\}^*$ be a binary encoded input to the program $\mathcal{P}$ and $P \subseteq E^*$ as before. We define* $\text{cfgpath}_{\mathcal{P}} : \{0,1\}^* \to P$ *to be a function that maps the input ι to the concrete execution path $p = (e_1, e_2, \dots)$. We consider a concrete execution path to be the edges taken through the CFG by a concrete state trace.*

Extending the definition for cfgpath to nondeterministic programs would be possible. This would mean that the same input ι can map to more than one execution path in the CFG. This can then result in different coverage profiles for the same input, depending on the randomness. As we will later see this is not beneficial for fuzzing purposes [LLV20], hence for the purposes of this thesis we assume that the program is deterministic. We can now define basic-block- and edge-coverage for an input to the program $\mathcal{P}$.

Definition 2.6 (Basic Block Coverage) *Let $\mathcal{P}$ be a deterministic program and $I = \{0,1\}^*$ the set of all possible binary encoded inputs. Then basic block coverage is defined as*

$$\begin{aligned} \text{bcov}_{\mathcal{P}} &: I \to C \\ \iota &\mapsto \text{pbcov}\,(\text{cfgpath}\,(\iota)) \end{aligned}$$

Definition 2.7 (Edge Coverage) *Let $\mathcal{P}$ be a deterministic program and $I = \{0,1\}^*$ the set of all possible binary encoded inputs. Then edge coverage is defined as*

$$\begin{aligned} \text{ecov}_{\mathcal{P}} &: I \to C \\ \iota &\mapsto \text{pecov}\,(\text{cfgpath}\,(\iota)) \end{aligned}$$

2.5.2 Coverage Feedback

As mentioned at the beginning of this section, fuzzers like AFL use coverage feedback in order to evolve a corpus. We will now discuss the used mechanism in more detail. Recall the state types introduced in Section 2.1. The *concrete state space*, i.e. the set of possible *concrete state traces*, can be mapped to an *abstract state space*. Coverage guided fuzzing does this by mapping the execution trace, i.e. a path, of a SUT to the coverage produced by this trace with the ecov function.

The ecov function is implemented by instrumenting the SUT. AFL for example uses a shared-memory region in order to share the coverage information between fuzzer and SUT. The fuzzer inserts startup code into the SUT that initializes this memory region. Each byte in the allocated region is mapped to one basic block in the program. The basic blocks are then changed to include an operation that increases the counter in the corresponding memory region when the basic block is executed.

After the SUT finishes executing, AFL can then read the values in the shared-memory region. AFL compares the coverage achieved by the input to the overall coverage already achieved. If a basic block was previously hit zero times and is now covered, the input is considered interesting and added to the seed corpus. The input is also added to the seed corpus, if a basic block was covered a greater amount of times than previously observed.

2.6 Systems under Test

This section gives an overview of different SUTs and their peculiarities. We will discuss the differences between hardware and software SUTs, and different types of software SUTs.

2.6.1 Hardware

Although fuzz testing usually tests software, it is possible that a user does not have direct access to the software in question. Instead of testing only the software, the alternative then is to test the hardware device in question and consider it as a blackbox. This situation where only hardware is available often arises when manufacturers integrate parts from other manufacturers. Verifying that the integrated parts are secure to a certain degree is required by the IEC 62443 standard, and the standard lists fuzzing as a possible testing measure [Com18]. This warrants the research done for blackbox fuzzing, since non-blackbox approaches are not applicable in this case. Further, although there seems to be no research considering this yet, it seems reasonable to extend fuzzing to actual hardware like integrated circuits.

2.6.2 Software

Software SUTs are currently the main interest in research. However, software SUTs can vary largely in their structure. As such, we will group them into different categories and point out major differences.

Firstly, there are SUTs that are only available as a binary. These binaries can still be instrumented retrospectively by modifying the assembly, at least enabling fuzzing with greybox approaches like AFL. Whitebox approaches however are not applicable, since they usually require knowledge about the original source code.

Secondly, we will consider software where the full source code is available, although the following categories are also applicable to software where only the binary is available. Fuzzing was originally created to fuzz command line tools like `grep`, `awk`, and many others. These SUTs all have in common, that they process an input read from a file or from standard input and then exit, once processing has finished. Such a "classic" SUT is depicted in Figure 2.5. Although such SUTs can have complex states in their program, the internal state only depends on a single input that is parsed. Highly complex SUTs for example include compilers like `gcc` or `clang`.

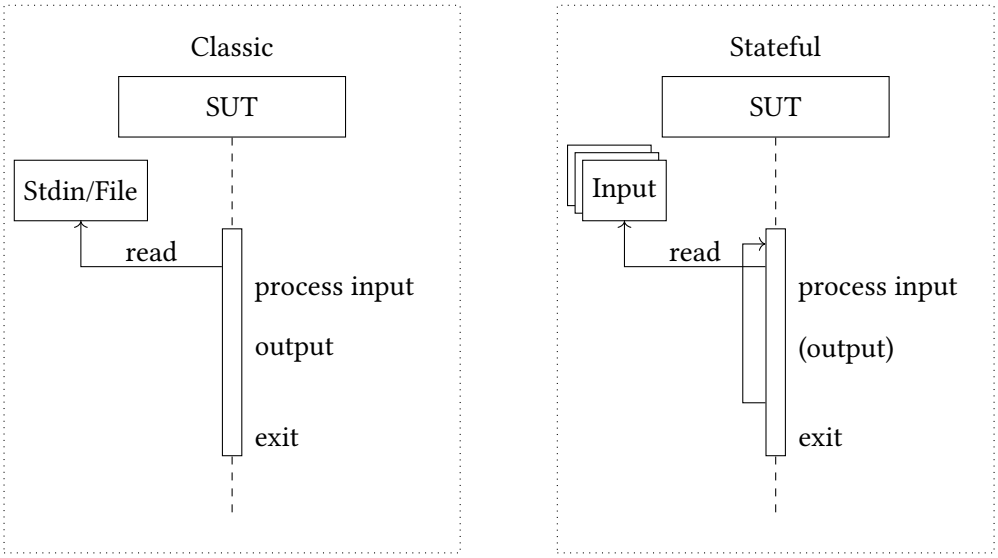


Figure 2.5: A "classic" SUT lifecycle (left) compared to a "stateful" SUT lifecycle (right).

We will thus consider "stateful" SUTs separately. The most notable difference between stateful and non-stateful SUTs is, that stateful SUTs do not exit after processing a single input as seen in Figure 2.5. Instead, they process a potentially infinite sequence of inputs that evolve

the internal state over time. The SUT waits for an input, and processes it once it is received. Afterwards, the SUT may optionally output something, before returning to waiting for new inputs. The SUT continues with this either until some condition is met, or until the SUT is killed by a user. As we will see in Chapter 3, this poses challenges, when applying classical fuzzing approaches like AFL directly.

3 Analysis

In this chapter we will take a closer look at the challenge of stateful fuzzing. We will first identify open issues of current approaches when fuzzing stateful software. Furthermore, we will take a look at the approach employed by AFLNet [PBR20b], which was able to improve fuzzing of stateful SUTs that implement protocols like FTP and RTSP. We will then identify possible improvements of the AFLNet algorithm.

3.1 Challenges of Stateful Fuzzing

Popular grey-box fuzzers like AFL [Zal+20], libFuzzer [LLV20], and honggfuzz [Goo20c] employ techniques that are effective when fuzzing SUTs that process an input and then exit. It is important to note the difference of states as described in Chapter 2. As we saw in Section 2.6, stateful SUTs usually process more than one input without exiting, posing challenges when applying existing techniques, since they usually test "classic" SUTs that exit after processing a single input. Although classic SUTs have internal state as well, this state does not persist across inputs and only depends on a single input and its structure. In contrast, since stateful SUTs process multiple inputs without exiting, they may have states that evolve over the course of processing multiple inputs. Hence, it makes sense to try to find abstraction functions (see Section 2.1) that are able to represent these states such that the fuzzer is able to use this information to its advantage. As we will see in Section 3.2, AFLNet uses the responses of the SUT in order to do so.

However, let us first consider the example protocol introduced in Section 2.3. In order to perform any meaningful operations, a client first has to establish a connection by sending an OPN message. Afterwards, for example a `DeleteNode` message can be sent. If there is no node to remove, this operation will fail, producing an ERR message. However, if previously a `CreateNode` message was sent, the `DeleteNode` command will exercise different behavior of the SUT than if no `CreateNode` message was sent. A sequence of inputs that may lead to interesting behavior of the SUT could thus for example be `OPN`, `CreateNode`, and finally `DeleteNode`.

For classic fuzzers like AFL that are tailored to classic SUTs the above sequence of messages is difficult to find. The fuzzer would first have to find three correct messages. Further, the fuzzer needs to figure out the correct order of the messages. Finally, the arguments of the messages have to be correct, i.e. the node id argument for `CreateNode` and `DeleteNode` has to be the same. AFL-like fuzzers usually employ random mutation of a set of input seeds. By using only random mutation of a single seed of input messages it is unlikely to find a sequence as the one described above. When the random mutations of the input produce no interesting new behavior of the SUT, fuzzers like AFL try to splice together existing seeds. This helps with finding message sequences. It is however still unlikely to find a correct sequence, because the fuzzer has no concept of a message. That is, the fuzzer does not know where to concatenate inputs in order to produce a correct sequence of inputs. In summary, there seem to be four main challenges in fuzzing stateful software. We will briefly present them and point towards possible solution approaches.

Seed Mutation A fuzzer that considers the stateful nature of a SUT has to perform seed mutation just like classic AFL-like fuzzers. However, in contrast to a classic fuzzer, the stateful fuzzer should incorporate knowledge about input structure into its mutations, in order to avoid wasting computing resources on inputs that are instantly rejected. Further, for stateful software sequences of inputs are relevant. This means the fuzzer needs to consider not only mutating individual inputs, but also the combination of inputs, i.e. it has to perform sequence mutations.

Furthermore, when trying to discover new messages, magic values can pose a problem for the mutation algorithm. Magic value refers to a value like the string `"HEL"`, that is used in a comparison against an input in order to trigger a specific branch in the SUT. Finding this value randomly has a probability of $1/2^{24}$. OPC UA for example has many more message types than the ones mentioned previously. By using only random mutations it is unlikely to find the identifiers needed to exercise new message types. This problem however is not exclusive to stateful fuzzing, since complex file types often also have magic values that determine the structure of the data contained in the file. There already exist several approaches to alleviate this, which can be applied to stateful mutations as well [laf16; Li+17; Asc+19].

Seed Selection Just like any other fuzzer, a stateful fuzzer has to select a seed to mutate. A stateful fuzzer could incorporate additional information into selecting seeds that promise to execute specific program states that appear to be more interesting than others.

State Identification An important part of a stateful fuzzer is the state identification. A stateful fuzzer needs a mechanism to identify states of the SUT, in order to guide the fuzzing process more effectively. As we have seen in Section 7.4, for blackbox fuzzing approaches this has already been extensively looked into [Dou+12; RP15; Fit+20; AGP19; Ma+16; Gas+15]. Blackbox approaches usually use the responses produced by the SUT and try to extract state information from the responses. This state information is then used to guide the fuzzer. Greybox fuzzers however can also benefit from more feedback information from the SUT as shown by Salls et al. [Sal+20].

State Selection Finally, a stateful fuzzer also needs to prioritize the fuzzing of the identified states. For example, consider the case that the state identification mechanism identified three states, disconnected, connected, and errored. It seems likely that the most progress can be made in the connected state, so the fuzzer would then continue fuzzing this state. However, since the concept of, for example, a connected state is domain specific knowledge depending on the SUT, the challenge is to find an abstraction that works reasonably well for most SUTs.

As discussed in Section 7.4, there currently are no approaches regarding fully automatic greybox fuzzing of stateful software that consider a state evolving over the course of multiple inputs. The closest approach to this is AFLNet by Pham et al. [PBR20b], that uses greybox fuzzing combined with state extraction to improve performance on stateful SUTs. Their approach however requires manual work in order to adapt new protocols. We will now discuss their approach in detail and identify possible improvements.

3.2 AFLNet

Pham et al. showed that their stateful fuzzer called AFLNet outperforms classic AFL and the state of the art blackbox fuzzer boofuzz [Per20] significantly on stateful SUTs [PBR20b]. They construct a state machine by using a protocol specific state extractor in order to guide the fuzzer in addition to coverage feedback. We will first present the results AFLNet achieved. In order to motivate possible improvements to AFLNet, we will then introduce AFLNet's architecture as described in the original paper. Finally, we will identify locations to improve the AFLNet algorithm.

3.2.1 Results

Pham et al. show that using stateful fuzzing significantly increases the coverage when compared to previous approaches. They evaluate their AFLNet fuzzer against the stateful blackbox fuzzer

boofuzz [Per20] and a modified AFL [Zal+20] (referred to as AFLNwe in the following) that can send the inputs over a network interface instead of standard input or a file. As SUTs, they use lightftp and live555, two open source implementations of the two popular internet protocols FTP and RTSP respectively.

		Branch Coverage			Statement Coverage		
		% Incr.	$\hat{A}_{12}$	p-value	% Incr.	$\hat{A}_{12}$	p-value
AFLNet vs AFLNwe	lightftp	121.06	1.000	< 0.001	79.45	1.000	< 0.001
	live555	3.40	0.335	0.076	2.44	0.228	0.003
AFLNet vs boofuzz	lightftp	57.73	1.000	0.026	49.72	1.000	0.026
	live555	64.13	1.000	0.026	62.09	1.000	0.026

Table 3.1: Mean coverage increase (%Increase), effect size ($\hat{A}_{12}$), and statistical significance (p-value) when comparing AFLNet to boofuzz and AFLNwe, respectively. A Vargha-Delaney $\hat{A}_{12}$ measure above 0.71 indicates a large effect size in favor of AFLNet. Statistical significance is computed using the Mann-Whitney U test (adapted from [PBR20b]).

In their experiments, boofuzz was supplied with a detailed model of the protocol, i.e. a state machine and message templates [PBR20b]. AFLNet and AFLNwe were only supplied with an initial seed corpus obtained from recordings of common message exchange scenarios [PBR20b]. Compared against boofuzz, AFLNet achieves a significant coverage increase as shown in Table 3.1. This shows that when using a less detailed model of the protocol (see Section 3.2.2) in combination with coverage feedback mechanisms as introduced in Section 2.5, a higher coverage can be achieved than by using a blackbox only approach. Furthermore, the large coverage increase for AFLNet compared to AFLNwe on lightftp suggests that the stateful components introduced by AFLNet increase effectiveness of greybox fuzzing. The authors explain the low coverage increase on live555 with a smaller state machine, i.e. the number of messages in a valid sequence is smaller [PBR20b]. They also argue that the number of functional states, i.e. states that are not error states, is smaller [PBR20b].

In general, these results suggest that incorporating state information to greybox fuzzing increases fuzzing effectiveness. Therefore, in the following section we will look at AFLNets architecture more closely and identify points to improve the existing approach.

3.2.2 Architecture

AFLNet extends the classic AFL fuzzing algorithm by introducing a *request sequence parser* component, a *state machine learning* component, a *target state selector* component, a *sequence selector* component, and a *sequence mutator* component as depicted in Figure 3.1.

The initial corpus in AFLNet consists of a set of seeds that contain a complete message, or a complete sequence of messages. The seeds are first split into individual requests by the *request sequence parser* component, before being further processed. This is done by implementing a function for each supported protocol which parses the protocol header and terminator in order to find the boundaries of individual requests. Each new message or sequence of messages added to the corpus is processed in the same way as the initial seeds.

AFLNet replaces the classic AFL mutation component by the *sequence mutator* component seen in Algorithm 2. This component first splits the request sequence of the currently processed seed into three parts M_1, M_2, and M_3. M_1 is a list of requests required to reach the currently selected state s. The candidate subsequence M_2 is a list of requests, that can be sent without leaving the state s. The rest of the requests is contained in M_3. The candidate subsequence M_2 is then mutated by either using traditional greybox mutation operations, or by using sequence mutation operations. These sequence mutation operations include insertion, duplication, substitution, or deletion of messages in M_2. The insertion and substitution operations choose a random request from a random seed and either insert it at the beginning or end of M_2, or replace M_2 with the new request.

Algorithm 2: Modified AFL fuzz function to incorporate Sequence mutations and the splitting of the sequence into M_1, M_2, and M_3.

```
1  def Fuzz(state, seed):
2      ⟨M1, M2, M3⟩ ← seed;
3      M' ← M2;
4      for MAX_HAVOC times do
5          for MAX_STACK times do
6              mutator ←random— AflMutators ∪ {Ins,Sub,Dup,Del};
7              M' ← mutator(M');
8          end
9          RunTarget(⟨M1, M', M3⟩);
10         M' ← M2;
11     end
12     return;
```

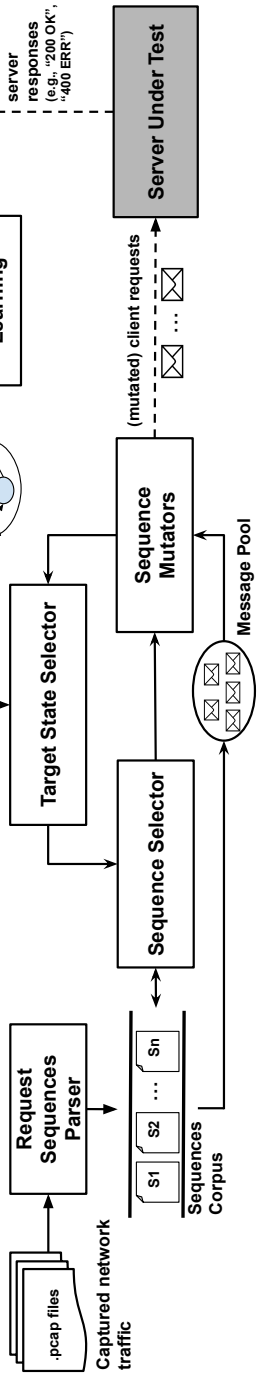


Figure 3.1: AFLNet architecture overview [PBR20b].

It should be noted that all mutation operations done in the inner loop starting in Line 5 stack on top of each other, except for the replacement mutator. The replacement mutator will discard any previous changes in favor of the randomly selected request. Once *MAX_STACK* mutations have been performed, the modified sequence will be sent in Line 9.

The *RunTarget* function in Line 9 starts the SUT and waits for it to initialize. It then sends the fuzzed input request by request, awaiting a response after each sent request. After all requests have been sent, the *state machine learning* component concatenates the received responses. It then parses them using a protocol specific response code extraction function in order to extract protocol specific response codes. Each new response code corresponds to a new state in the state machine. If a previously observed response code is observed, an edge is added to the state machine if it does not exist yet.

The remaining two components, the *target state selector* and *sequence selector*, each use three different selection strategies. For both components, a random and a round-robin selection strategy is supported. The random strategy simply chooses a state or sequence uniformly at random. The round-robin strategy cycles through the states or sequences respectively. Both selector components support a third strategy that uses a scoring mechanism. These scoring strategies are discussed in the following.

Algorithm 3: AFLNets weighted state selection strategy.

```
1 def ChooseStateFavored():
2     foreach state ∈ States do
3         state.score ← ⌈1000 · 2^(−log10[log10(S_f+1)S_s+1] 2^log(S_p+1))⌉;
          // S_f is the number of times the state was fuzzed
          // S_s is the number of times the state was selected as a target
          // S_p is the amount of new paths that were discovered during
             fuzzing with the state as target
4     end
      // Now select a state at random,
      // with probabilities based on the scores
5     ∀s ∈ states : p_s ≔ s.score / Σ_{s'∈states} s'.score;
6     state ←(with p_s)— states;
7     return state;
```

The *target state selector* scoring function takes into account how often a state was fuzzed, how often it was selected, and how many new discoveries were made while fuzzing the state. The function described in Algorithm 3 first calculates a score for each state. It then assigns a

probability to each state and randomly chooses a sample from the set of states according to the calculated probabilities. A high score corresponds to a high probability to be chosen.

The AFLNet authors reason that in order to identify fuzzer blind spots, rarely exercised states are assigned a probability to be chosen inversely proportional to the number of times a sequence was fuzzed in the state and inversely proportional to the number of times the state was selected [PBR20b]. This is reflected by the term $2^{-\log_{10}(\log_{10}(S_f+1)S_s+1)}$ in the scoring function in Algorithm 3. In order to increase the probability of discovering interesting inputs that produce new state transitions or new coverage, the score of a state is also proportional to the number of discoveries made in the state [PBR20b]. This is reflected by the term $2^{\log(S_p+1)}$ in the scoring function in Algorithm 3. The constant factor of 1000 is an implementation detail and not important for the overall distribution of probabilities, since it cancels out when calculating the individual probabilities in Line 5.

The *sequence selector* scoring uses a slightly adapted scoring mechanism from traditional AFL. We will not discuss the AFL seed selection mechanism [Zal+20] in detail here but only highlight two major changes made by AFLNet to the selection strategy. Firstly, in contrast to AFL, the seed is selected such that it reaches the selected state at least once. That is, seeds that are not able to reach the currently selected state are skipped entirely. Secondly, seeds are skipped with probability 0.9 if they were generated in a different state than the currently selected one. If there are less than ten seeds available that reach the selected state, the seeds are selected in a round-robin fashion.

The overall control flow of the selection components is displayed in Algorithm 4. As long as the fuzzer is not signaled to exit, it will select a state according to the previously discussed state selection algorithm. Then the fuzzer calls the cull queue function analogously to AFL (see Section 2.5). Afterwards it will select a seed according to the previously discussed seed selection strategies. The fuzzer will then proceed to fuzz the selected seed in the selected state as described in Algorithm 2.

Algorithm 4: AFLNet main loop

```
1 while not exit do
2 |   state ← ChooseState();
3 |   CullQueue();
4 |   seed ← ChooseSeed(state);
5 |   Fuzz(state, seed);
6 end
```

3.2.3 Flaky Coverage

AFLNet executes the program and simply sends data to it and then tries to receive data, waiting for a specified maximum timeout. This poses a problem for fast SUTs, since the main loop can run a different amount of times on the same input sequence. For example, if AFLNet is slow during the current fuzzing iteration, the SUT might be able to run its main loop twice. Another time, AFLNet might be faster, and the main loop might then only be run once.

This is problematic, since it causes the coverage to differ for the same input sequence. This nondeterministic behavior also occurs, if the input is slightly mutated, but does not produce any actual different behavior. If it just loops a different amount because it had more time, it will be added to the corpus regardless. In order to alleviate this problem, the SUT loop will have to be synchronised with the fuzzer loop. This is discussed in more detail in Section 4.4.

3.2.4 Limitations and Possible Improvements

We now revisit each component of AFLNet. For each component we will briefly discuss current limitations and possible improvements. In Section 3.3 we will then discuss which of these should be handled first, and pose corresponding research questions.

Request Sequence Parser

Recall from Section 3.2.2 that the *request sequence parser* requires specification in order to extract message boundaries. Even though the protocol does not have to be fully implemented in order to extract message boundaries, it is still manual work that has to be performed in order to be able to fuzz SUTs that have no such implementation yet [PBR20b]. Reducing the amount of work a user needs to perform in order to start fuzzing, will most likely increase the rate of adaption. There already exist approaches that try to automatically infer the message format of a protocol, which could be applied to the *request sequence parser*, omitting the need for manual specification [FC18; PP16; Cab+07; Cui+08].

State Machine Learning

Similarly, the *state machine learning* component needs a specification of the protocol's messages, in order to be able to extract the response codes. The previously mentioned automatic message inference mechanisms could also be applied to the *state machine learning* component. However, this only alleviates the problem of manual specification. As the authors of AFLNet also identified, the response codes in some cases do not convey sufficient state information in order to improve fuzzer effectiveness [PBR20b]. One approach to alleviate this is to modify the source

code to extend the response codes, such that they convey more state information [PBR20a]. Nevertheless, this again requires manual work and an understanding of the SUTs source code when adapting a new SUT. A different approach to extract state information when using greybox fuzzing could be to use the already provided coverage feedback.

Sequence Mutator

The *sequence mutator* component currently only considers random requests from the entire corpus for its operations. Böhme et al. recently proposed using the concept of entropy for a seed to more efficiently guide libFuzzer's fuzzing process [BMC20]. They define a seed's entropy roughly as the amount of information that is gained by executing the program on this seed. By using this concept they were able to achieve a substantially improved efficiency of the fuzzer [BMC20]. This concept could be applied to the sequence mutations as well. Instead of the sequence mutator choosing the request to insert uniformly at random, it seems plausible that selecting requests with high entropy should yield better results. The same reasoning can be applied to the *sequence selector* component. However, in this case the proposed entropy based scheduling could be directly applied. Another possibility could be to extend the calculation of the entropy to include the additional information gained by extracting states.

State Selection

Finally, the *state selection* component can be adjusted. The current state weighting function only considers the amount of times the state was fuzzed and selected, and the number of discoveries made in this state. Furthermore, except for their proportionality the choice of how to put these values in relation seems arbitrary. Possible improvements could thus include finding additional metrics, or a better weighting function choice.

3.3 Research Questions

Making fuzzing more accessible to more people should always be considered a high priority. Because if more developers fuzz their software, more bugs can be found, than if the software is not fuzzed at all. In order to increase adaptation of AFLNet it is worth investigating automated approaches for the components that currently need manual work in order to adapt new SUTs. This is why we will only consider the *request sequence parser* and *state machine learning* component. However, since automatic message inference has already been investigated by others (see Section 3.2.4), we will focus on the *state machine learning* component for this thesis. Also, there exists a lot of software especially in the embedded sector that does not provide

much feedback in responses. This makes extraction of information difficult, when considering only the responses sent by the SUT. Because of this, we will try to use the existing coverage feedback instead of the responses supplied by the SUT.

Furthermore, because of the stateful nature of the SUTs we consider, it seems plausible that using a mechanism to snapshot the server in a specific state and continue from that point onward, could speed up the fuzzing process. This is relevant, because fuzzers like AFL or libFuzzer are still an order of magnitude faster than AFLNet. We will thus investigate if a simple snapshotting mechanism can provide a significant performance boost.

Finally, Klees et al. [Kle+18] mention that the initial corpus can have a significant effect on the results of the fuzzing campaign. We will investigate the effect this has on stateful fuzzing. Thus, we pose the following four research questions:

RQ1 Can the state identification process be automated by using already provided coverage information?

RQ2 How effective is a fuzzer using an automated state identification process compared to the manual approach of AFLNet if measuring performance by using coverage achieved as metric?

RQ3 Does the seed corpus selection make a difference in achieved coverage when fuzzing stateful software?

RQ4 Could program state snapshotting improve fuzzing speed and as such effectiveness?

4 Methods

In this chapter, we introduce a novel algorithm that can be used to automatically identify states of an SUT. First, an overview of all changes to AFLNet's components is provided in Section 4.1. We afterwards discuss the introduced techniques in detail, starting with the definition of the concept of message behavior needed to describe the algorithm in Section 4.2. Using the concept of message behavior, we motivate the state identification algorithm and then formally define it in Section 4.3. In Section 4.4, we then introduce a synchronisation mechanism, in order to make the SUT execution more deterministic. This is required in order for the fuzzer to be able to extract the message behavior. Furthermore, we shortly discuss how AFLNets mutation strategy was modified. Finally, the concept of a state forkserver is introduced and briefly described.

4.1 Overview

Recall that the objective of this thesis (as motivated in Section 3.3) is to automate AFLNet's *state machine learning* component. Figure 4.1 depicts the changes we introduce in this chapter, in order to achieve this objective. Components highlighted in cyan are only slightly modified, whereas components highlighted in green are either replaced, or entirely new.

The biggest change consists of the new *state machine learning* component, which replaces the current component of AFLNet. The component in essence builds a mealy automaton that models the SUT's state transitions, when provided with a sequence of inputs and a final message. This approach is described in detail in Section 4.3. However, in order for the approach to work, the concept of "message behavior" (introduced in Section 4.2) is required. We use the message behavior as a replacement for the response code extraction employed by AFLNet and as such eliminate the necessity for manual specification. Furthermore, we introduce a new *synchronisation component* that is responsible for making the SUT's execution deterministic, which is required for the message behavior extraction. Finally, we slightly modify the existing *sequence mutator* component of AFLNet, such that it only mutates the last message, since this will be a requirement for our algorithm to work correctly. In addition, the probability with which sequence mutations are performed is changed to depend on information obtained during state identification.

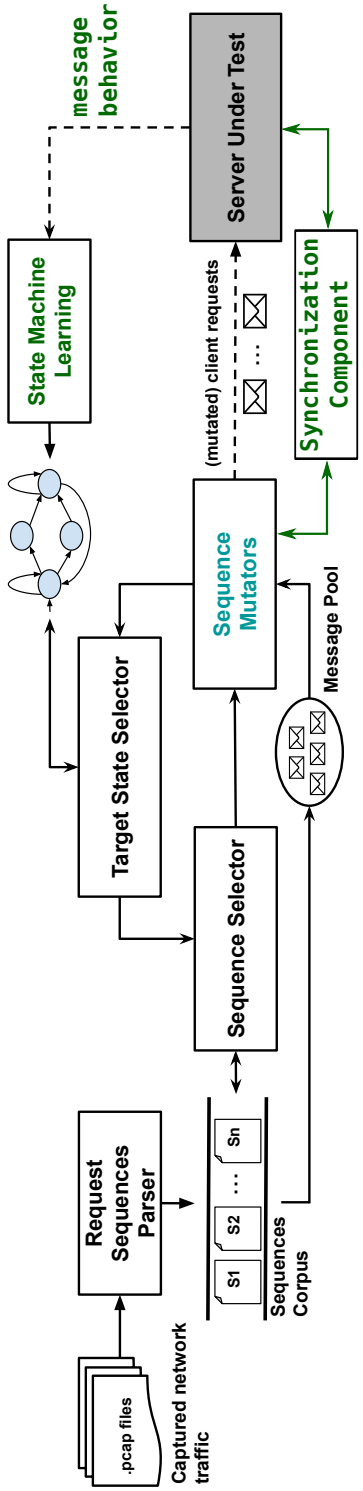


Figure 4.1: AFLNet architecture overview with changes highlighted in cyan and additions highlighted in green.

4.2 Message Behavior

In this section we define the concept of "message behavior". This is later needed in order to describe the newly devised state identification algorithm. Recall that AFLNet uses a *request sequence parser* component in order to extract requests from a corpus input file. For now, we call multiple requests a sequence, and a request sent after such a sequence a message. We later formally define messages and sequences.

Let us now recall the example program from Section 2.3 (Listing 4.1) and use it as SUT. Consider the three different sequences of requests in Table 4.1 and the coverage that the message sent after the sequence produces on the SUT.

```
1  connected = False
2  while True:
3      msg = recv(timeout=0.05)
4      if msg.typ == "OPN":
5          response = do_opn(msg)
6          connected = True
7      elif msg.typ == "MSG" and connected:
8          response = do_msg(msg)
9      elif msg.typ == "CLO" and connected:
10         response = do_msg(msg)
11         connected = False
12     elif msg is not None:
13         response = "ERR"
14     if response is not None:
15         send_response(response)
16     do_background_processing()
```

Listing 4.1: Example Program based on OPC UA, modelling a simple server.

Similar to Definition 2.6, the coverage in Table 4.1 is a tuple of values that correspond to how often each line was executed, i.e. the first element in the tuple corresponds to how often the first line was executed and so on. Instead of looking at the coverage of the whole execution of the SUT, we will consider the coverage caused by the execution of a message sent after a sequence and call this the message's behavior. For example, for the first sequence-message pair the sequence is empty, and we only send the message MSG. The SUT's execution on this sequence-message pair leads to the coverage in the first row of Table 4.1. In this case, the coverage of the entire execution of the SUT corresponds to the coverage the message would produce, i.e. its behavior. The same holds for the second sequence-message pair.

Sequence	Message	Coverage
⟨⟩	MSG	(1,1,1,1,0,0,1,0,1,0,0,1,1,1,1,1)
⟨⟩	OPN	(1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1)
⟨OPN⟩	MSG	(1,2,2,2,1,1,1,1,0,0,0,0,0,2,2,2)

Table 4.1: Sequences and the coverage they produce on the program in Listing 4.1.

In the last sequence-message pair, the sequence consists of the OPN request, and the message is a MSG request. In order to get the coverage the message produces, the coverage after the sequence, which only consists of the OPN request, (1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1) can be subtracted elementwise from the final coverage observed after the sequence-message pair has been completely processed (1,2,2,2,1,1,1,1,0,0,0,0,0,2,2,2). The coverage, i.e. the behavior, of the message MSG, then is (0,1,1,1,0,0,1,1,0,0,0,0,0,1,1,1). The resulting behavior when sending MSG after OPN is different from the behavior observed when sending MSG alone. From this differing behavior we can conclude that the program must have modified its internal state. We use this insight to devise an algorithm that infers a state machine using this concept of message behavior.

Before introducing the state machine algorithm, we formally define message behavior. First, we extend Definition 2.6 and Definition 2.7 for messages and message sequences.

Definition 4.1 (Message Sequence) *Let $\mathcal{P}$ be the executable of the SUT and $I = \{0,1\}^*$ the input space of $\mathcal{P}$. Let $\chi_{\mathcal{P}} : I_{\mathcal{P}} \to \{0,1\}$ be a function depending on the executable $\mathcal{P}$ that outputs 1 if an input can be further processed by $\mathcal{P}$ and 0 otherwise. We then call $M = \{\iota \mid \iota \in I : \chi(\iota) = 1\} \subseteq I$ the set of messages $\mathcal{P}$ accepts.*

We interpret an arbitrary length bit string $s \in I$ as a sequence of messages $s = (m_1, m_2, \dots)$. If $|s| < \infty$ it is possible that the last bits cannot be classified as a message. We will call these bits an incomplete message.

For network protocols, the function χ usually uses something simple like a fixed value, or some kind of delimiter to determine when a message is complete. The FTP protocol for example separates the input into messages by using the sequence `\r\n` as delimiter. Another often occurring input separation function is a fixed width header, which specifies the length of the body.[1] The function χ would in that case output 1 if the input contains the header, and the amount of bits specified in the header.

With Definition 4.1 and Definition 2.7 we can now define the behavior of a message in context of a previous sequence of inputs. For this definition, we will only consider finite sequences.

1 A notable example is the IP protocol

An infinite sequence would never allow a last message to be applied to the program, hence rendering the concept useless.

Definition 4.2 (Message Behavior) *Let $\mathcal{P}$ be the executable of the SUT and C the set of possible coverage profiles for $\mathcal{P}$. Further, let M and I be as in Definition 4.1. We define the behavior of the program $\mathcal{P}$ for a message m after a sequence of previous messages $s = \langle m_1, \ldots, m_n \rangle$ as*

$$\begin{aligned} \text{beh}_{\mathcal{P}} \colon M^* \times I &\to C \\ (s,m) &\mapsto \text{ecov}\left(\langle m_1, \ldots, m_n, m \rangle\right) - \text{ecov}\left(\langle m_1, \ldots, m_n \rangle\right) \end{aligned}$$

where the subtraction of the coverage profiles is performed elementwise on the tuples.

In essence, Definition 4.2 defines the behavior of the program for a message m after a sequence s to be the coverage profile of the message m, after the program has processed the messages of the sequence s. As noted in the exemplary discussion at the beginning of this section, a crucial observation is that Definition 4.2 implies that if $\text{beh}_{\mathcal{P}}(s_1, m) \neq \text{beh}_{\mathcal{P}}(s_2, m)$ for message m and sequences s_1 and s_2 with $s_1 \neq s_2$, we must have observed some kind of state change of the program. Otherwise, the message m would have produced the same behavior, as long as the SUT is deterministic. This leads us to the algorithm described in the following section.

4.3 State Identification Algorithm

We will now describe the devised algorithm to identify different program states. The devised approach does not directly identify states that correspond to a protocol state. Rather, we try to identify a mealy automaton that models the behavior of the SUT. Sending a message corresponds to traversing an edge in the SUT's automaton. Since we model a mealy automaton, traversing an edge produces an output, which in our case is the behavior of the message sent. We construct this automaton by sending inputs to the SUT and observing the behavior they exhibit. If we find a message for which the observed behavior contradicts our already constructed state machine model, the conflict is solved by changing the model accordingly. A contradiction occurs, if a message exhibits two different behaviors when sent in the same state. We now use the example protocol introduced in Section 2.3 (repeated in Listing 4.1) to motivate the algorithm, before describing it formally.

Each state consists of two sets that are used by the algorithm as depicted in Figure 4.3. The first set $\mathcal{S}$ contains the sequences of messages that are able to reach this state, i.e. the set of paths through the state machine that reach the state. The second set $\mathcal{B}$ contains the behaviors

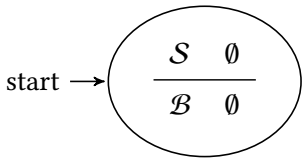


Figure 4.2: Initial state of the state machine.

observed when sending messages in the state. Each entry consists of a set of sequences (the *context set*) that were already sent in combination with the message, and the message's behavior, provided one of the sequences in the context was sent in order to reach the state. As previously mentioned, the edges of the state machine correspond to sending a message and observing the message's behavior as output. It should be noted, that the output depends on the path taken through the state machine. Also, the edges can be inferred from the sets $\mathcal{S}$ and $\mathcal{B}$. For the examples, we will explicitly model the edges in the figures. Later on in the formal description, we will omit the edges.

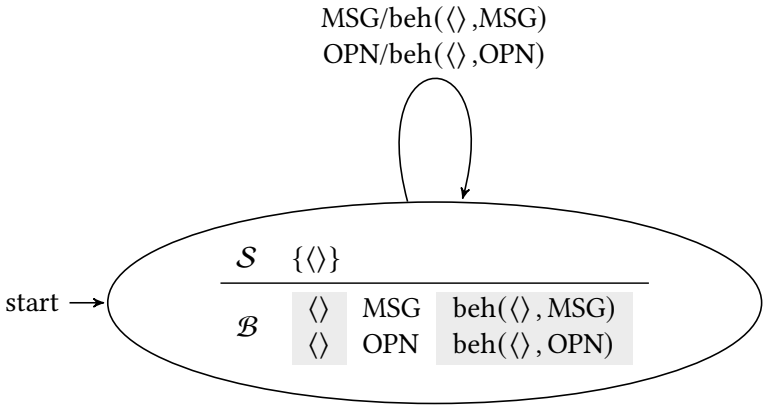


Figure 4.3: State of the state machine after observing MSG and OPN messages.

We start with an empty state machine as depicted in Figure 4.2. The set of sequences $\mathcal{S}$ that are able to reach this state is initially empty. The set of behaviors observed when sending messages in the state $\mathcal{B}$ is also empty, since we did not send any inputs yet.

Now consider the case that we send a MSG message to the SUT. We simply add the message to the current state, since this is the first message we send. This produces a self loop as seen in Figure 4.3. In order to add the message, we add a new behavior entry to $\mathcal{B}$. Since we only sent a single message, we add the empty sequence to the *context set* of the message. We add the message MSG as the second value of the behavior entry. We set the last value of the behavior entry to the behavior of the message provided that the empty sequence was first sent.

If we then send an OPN message to the SUT, we encounter a different case. Since now the empty sequence was already sent previously, we look for a state that is reachable by this sequence. In this case, this is the initial state. The way we construct our automaton in the following guarantees that a sequence reaches exactly one state, and as such identifies a state. Since we never sent the message OPN before, we add a behavior entry in the set of behaviors. This is also depicted in Figure 4.3.

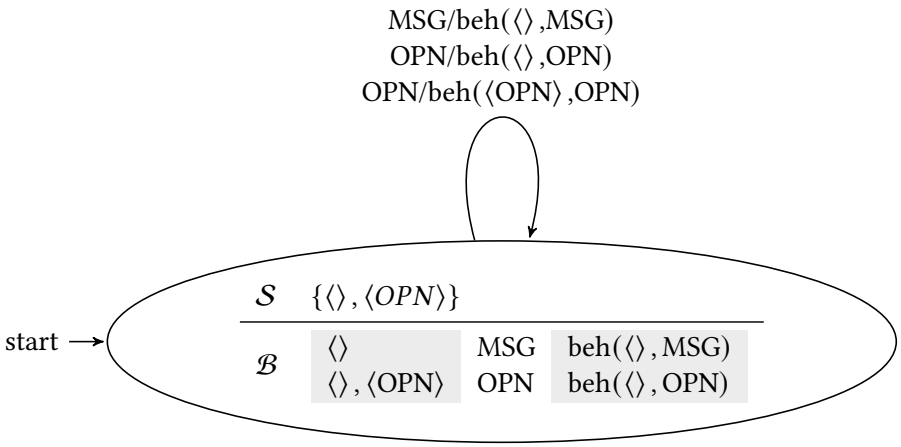


Figure 4.4: State of the state machine after observing the sequence ⟨OPN⟩ with message OPN.

The interesting cases occur, when we send a message that we previously sent. In this case it is possible that we observe a different behavior if we send the message with a different sequence than previously. Consider sending the sequence ⟨OPN⟩ and message OPN. Since the example program behaves the same no matter how many times an OPN message is sent, the behavior does not change, i.e. beh(⟨⟩, OPN) = beh(⟨OPN⟩, OPN). In this case we simply add the new sequence to the set of sequences $\mathcal{S}$ that are able to reach the state, and to the context of the behavior of the OPN message as depicted in Figure 4.4. This corresponds to adding the new edge OPN/beh(⟨OPN⟩,OPN).

Now consider sending the sequence ⟨*OPN*⟩ and the message MSG. For this sequence the behavior will be different, since the `connected` variable is now set to true by the OPN message. The MSG message therefore exercises a different behavior than if it were sent with an empty sequence. We cannot add the message behavior as previously, since that would produce contradicting outputs along the existing edge for the input MSG. If we added the edge, this would result in two edges with the same input but different outputs, resulting in a state machine that is nondeterministic, and as such not being a mealy automaton. In order to alleviate this, we modify the state machine as seen in Figure 4.5. We first remove the sequence ⟨OPN⟩ from the state's $\mathcal{S}$ set and move it to a newly created state, since the new state can be reached by

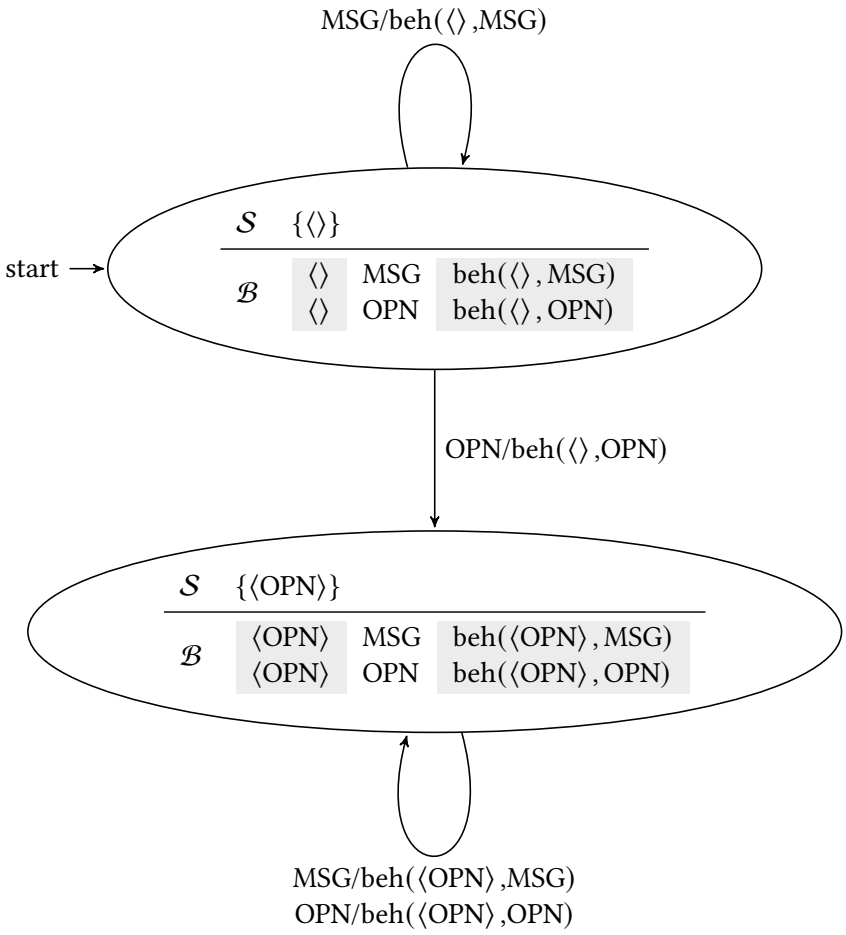


Figure 4.5: State of the state machine after observing the sequence ⟨OPN⟩ with message MSG.

this sequence. In order to reflect this, we redirect the edge with the OPN message with the behavior that was observed with an empty sequence to the new state, since we found that sending an OPN message produces a contradiction, and as such indicates that sending the OPN message changes the state of the SUT. The edge with the OPN message that was observed with the sequence ⟨OPN⟩ is moved to the new state, and we also add an edge for the current observation to the new state. This is also reflected in the behavior set $\mathcal{B}$.

Finally, consider the state machine to be as seen in Figure 4.3. If we then send the sequence $\langle OPN \rangle$ and the message MSG at this point in time, we will still observe contradicting behavior, since the behavior of the MSG message will differ from sending it after the sequence ⟨⟩. Adding the edge to the state machine would again yield an invalid mealy automaton because it would introduce nondeterministic transitions, i.e. the same input would produce different outputs

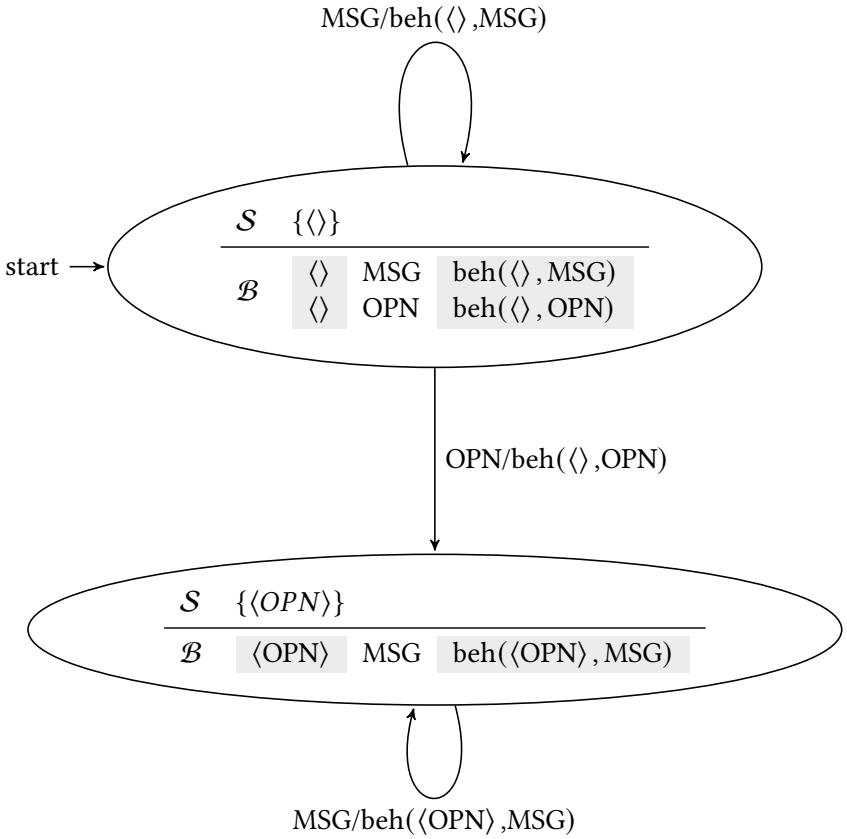


Figure 4.6: State of the state machine after observing the sequence ⟨OPN⟩ with message MSG.

on the same edge. In contrast to the previous case however, the sequence will be new. In this case we look for the state where the contradiction occurs and create a new state containing the newly sent sequence and behavior as seen in Figure 4.6. The edge of the OPN message is redirected like in the case seen in Figure 4.5.

After introducing the algorithm on a high level via the provided examples, we now formally define the algorithm. In order to do so, we first define our notion of a state.

Definition 4.3 (State) *A state* $\mathfrak{s} = (\mathcal{S}, \mathcal{B})$ *is a tuple of a set of already seen sequences* $\mathcal{S} = \{s_1 = (m_1, \dots, m_n), \dots\}$ *and a set of observed behaviors* $\mathcal{B} = \{\mathfrak{b}_1, \dots\}$*. An observed behavior* $\mathfrak{b} \in \mathcal{B}$ *is a 3-tuple* $\mathfrak{b} = (\mathfrak{c} = \{s_1, \dots\}, m, \text{beh}_{\mathcal{P}}(s,m))$ *consisting of a set of sequences* $\mathfrak{c}$*, a message* m *and the observed behavior of* m *given a sequence* $s \in \mathfrak{c}$*. We will call* $\mathfrak{c}$ *the context in which a message behavior was already observed.*

In order to keep the notation simple, for the following description of the algorithm we assume that elements are mutable. For example, an update of $\mathcal{S}$ in the form of $\mathcal{S} \leftarrow \mathcal{S} \cup \{s'\}$

means that the set was modified. This modification will be visible in $\mathfrak{S}$ without reassigning anything. Also, in order to denote access to a tuple member at position i, we will write $x^{(i)}$ for some tuple x. In contrast to arrays in most programming languages, we will start indexing from 1 instead of from 0, i.e. the first element of the tuple is $x^{(1)}$.

With Definition 4.3 we can now describe the algorithm to construct a state machine during fuzzing of the SUT executable $\mathcal{P}$. The algorithms initial state is described in Algorithm 5. We start with one initial state $\mathfrak{s}_1$. The sequences are initially empty, and the observed behavior set is empty as well. The two helper sets σ and μ are used to determine if a sequence or message respectively has been sent previously. Both σ and μ are initially empty.

For the following we will consider input $\iota \in I$ that was generated by the fuzzer and sent to the target $\mathcal{P}$. The input ι can be interpreted as a sequence of messages (see Definition 4.1) $\hat{s} = \left(m'_1, \ldots, m'_n, m'\right)$ of length $n + 1$. For the algorithm we will consider the prefix sequence $s' = \left(m'_1, \ldots, m'_n\right)$ and the last message m'. For the description of the algorithm we also use the concept of a current state. The current state refers to the state the fuzzer is currently using to generate messages from, i.e. the fuzzer sends a sequence that reaches the state, and then sends one or more mutated messages. If the fuzzer only sends one mutated message, the sequence will still be known. However, if the fuzzer sends more than one mutated message, the sequence will get longer, resulting in possible state transitions that the algorithm will identify.

Algorithm 5: Statemachine initialization.

```
1 𝔰₁ ← (∅, ∅) // Initial state
2 𝔖 ← {𝔰₁} // Set of states
3 σ ← ∅ // Set of already sent sequences
4 μ ← ∅ // Set of already sent messages
```

The state machine update behavior is described in Algorithm 6. The algorithm checks whether the fuzzer has sent the sequence s' or the message m' previously by performing a lookup in the sets σ and μ respectively. This results in the four cases in Lines 2, 4, 6 and 8 which we will discuss in detail in the following paragraphs. After handling the appropriate case, the sequence s' and message m' are simply added to the helper sets σ and μ respectively for future lookups in Lines 11 and 12.

The first case of the update algorithm is described in Algorithm 7. In this case the fuzzer sent a sequence and message that were never sent before. Since this is the first time the sequence s' and message m' are sent, there will be no conflicting behavior. Hence, we insert both the sequence and the message's behavior into the currently selected state $\mathfrak{s}$.

Algorithm 6: Statemachine update.

Input : The set of observed sequences σ, the set of observed messages μ, the set of states $\mathfrak{S}$, the current state $\mathfrak{s}$, the currently observed sequence s' and the currently observed message m'

```
1  def StateMachineUpdate(σ, μ, 𝔖, 𝔰, s', m'):
2      if s' ∉ σ ∧ m' ∉ μ then      // new sequence and new message
3          𝔖 ←CaseNewSNewM(𝔖, 𝔰, s', m');
4      else if s' ∈ σ ∧ m' ∉ μ then// already observed sequence and new message
5          𝔖 ←CaseOldSNewM(𝔖, 𝔰, s', m');
6      else if s' ∉ σ ∧ m' ∈ μ then// new sequence and already observed message
7          𝔖 ←CaseNewSOldM(𝔖, 𝔰, s', m');
8      else if s' ∈ σ ∧ m' ∈ μ then // already observed sequence and message
9          𝔖 ←CaseOldSOldM(𝔖, 𝔰, s', m');
10     end
11     σ ← σ ∪ {s'};
12     μ ← μ ∪ {m'};
13     return σ, μ, 𝔖
```

Algorithm 7: Statemachine update case: new sequence and new message.

Input : The set of states $\mathfrak{S}$, the current state $\mathfrak{s}$, the currently observed sequence s' and the currently observed message m'

```
1  def CaseNewSNewM(𝔖, 𝔰, s', m'):
2      (𝒮,ℬ) ← 𝔰;
3      𝒮 ← 𝒮 ∪ {s'};
4      𝔟' ← ({s'}, m', beh_𝒫 (s',m'));
5      ℬ ← ℬ ∪ {𝔟'};
6      return 𝔖
```

The second case of the update algorithm is described in Algorithm 8. Similar to Algorithm 7, since this is the first time the fuzzer sent message m', there cannot be a different behavior. However, since the sequence s' was previously sent, it has to be contained in some sequence set $\mathcal{S}$ of a state $\mathfrak{s} = (\mathcal{S},\mathcal{B})$. The algorithm searches for that state, i.e. the state that is reachable by the sequence s', and creates a new behavior entry $\mathfrak{b}'$ for the newly sent message and adds it to the set of observed behaviors $\mathcal{B}$. By construction of the state machine, since we model a deterministic mealy automaton, each state is uniquely identified by one or more sequences, i.e. each sequence is contained in at most one state.

The last two of the four cases in Algorithm 6 are the interesting ones, since the fuzzer will now have sent the message in a previous iteration. The algorithm now determines if the

Algorithm 8: Statemachine update case: observed sequence and new message.

Input : The set of states $\mathfrak{S}$, the current state $\mathfrak{s}$, the currently observed sequence s' and the currently observed message m'

```
1 def CaseOldSNewM(𝔖, 𝔰, s′, m′):
2     𝔰′ ← (𝒮,ℬ) ∈ 𝔖 with s′ ∈ 𝒮;
3     𝔟′ ← ({s′}, m′, beh_𝒫 (s′,m′));
4     ℬ ← ℬ ∪ {𝔟′};
5     return 𝔖
```

message was already seen in a different sequence context. As previously discussed with the example protocol, the idea then is to restructure the state machine, if the sequence produces a behavior of the message that conflicts with the already seen behavior.

Algorithm 9: Statemachine update case: new sequence and observed message.

Input : The set of states $\mathfrak{S}$, the current state $\mathfrak{s}$, the currently observed sequence s' and the currently observed message m'

```
1  def CaseNewSOldM(𝔖, 𝔰, s′, m′):
2      (𝒮,ℬ) ← 𝔰;
3      if ∃𝔰′ = (𝒮′, ℬ′) ∈ 𝔖 : ∃𝔟 ∈ ℬ′ : (m′ = 𝔟^(2) ∧ beh_𝒫 (s′,m′) ≠ 𝔟^(3)) then
4          S̃ ← {s′};
5          b̃ ← ({s′}, m′, beh_𝒫 (s′,m′));
6          B̃ ← {b̃};
7          s̃ ← (S̃, B̃);
8          𝔖 ← 𝔖 ∪ s̃;
9      else
10         𝒮 ← 𝒮 ∪ {s′};
11     end
12     return 𝔖
```

The case handling a new sequence and a previously seen message is described in Algorithm 9. The algorithm first checks whether there exists a state $\mathfrak{s}'$ such that it contains an observed behavior $\mathfrak{b}$ to message m' that has a conflicting message behavior (Line 3). That is, the already observed behavior $\mathfrak{b}^{(3)}$ is different from the currently observed one $\text{beh}_{\mathcal{P}}(s',m')$. If this is the case, the sequence s' must have triggered some kind of state change in the program. Otherwise, the behavior (i.e. the code that m' triggered the execution of) would not have changed. The algorithm reflects this by creating a new state $\tilde{\mathfrak{s}}$ and adding the sequence s' to the set of sequences for this state (Line 4). Then, the algorithm creates a new observed behavior for sequence s' and message m' and adds it to the state (Lines 5 and 6). Finally, the state is added

to the set of states $\mathfrak{S}$ (Line 8). If there is no state that satisfies the condition, the fuzzer adds the newly sent sequence s' to the set of sequences of the current selected state $\mathfrak{s}$ (Line 10).

Algorithm 10: Statemachine update case: observed sequence and observed message.

Input : The set of states $\mathfrak{S}$, the current state $\mathfrak{s}$, the currently observed sequence s' and the currently observed message m'

```
1  def CaseOldSOldM(𝔖, 𝔰, s′, m′):
2      𝔰′ ← (𝒮,ℬ) ∈ 𝔖 with s′ ∈ 𝒮;
3      if ∃𝔟 ∈ ℬ : (s′ ∉ 𝔟^(1) ∧ m′ = 𝔟^(2) ∧ beh_P(s′,m′) ≠ 𝔟^(3)) ∧ |𝒮| > 1 then
4          S̃ ← {s′};
5          b̃ ← ({s′}, m′, beh_P(s′,m′));
6          B̃ ← {b̃} ∪ {({s′}, 𝔟^(2), 𝔟^(3)) | 𝔟 ∈ ℬ : s′ ∈ 𝔟^(1)};
7          s̃ ← (S̃, B̃);
8          𝒮 ← 𝒮 \ {s′};
9          𝔖 ← 𝔖 ∪ s̃;
10         ℬ ← {(𝔟^(1) \ {s′}, 𝔟^(2), 𝔟^(3)) | 𝔟 ∈ ℬ : 𝔟^(1) \ {s′} ≠ ∅}
11     else
12         if ∃𝔟 ∈ ℬ : (m′ = 𝔟^(2) ∧ beh_P(s′,m′) = 𝔟^(3)) then
13             𝔟^(1) ← 𝔟^(1) ∪ {s′};
14         else
15             𝔟 ← ({s′}, m′, beh_P(s′,m′));
16             ℬ ← ℬ ∪ {𝔟};
17         end
18     end
19     return 𝔖
```

Finally, Algorithm 10 describes the case when dealing with both a previously sent sequence, and a previously sent message. First, the algorithm selects the state $\mathfrak{s}'$ which was previously observed to be reachable by the sequence s'. Afterwards, the algorithm performs a check in order to determine if the state $\mathfrak{s}'$ needs to be split such that a valid mealy automaton is obtained. Similar to Algorithm 9, the condition in Line 3 checks whether a conflicting behavior to message m' was observed. Only the behaviors for the state $\mathfrak{s}'$ are considered, since the algorithm previously determined that the sequence s' is able to reach this state. The algorithm searches the set of observed behaviors for an observed behavior for the message m' with differing message behavior $\text{beh}_{\mathcal{P}}(s',m')$. In addition, the algorithm requires that the sequence was not previously sent in combination with this message. For deterministic programs this condition is not necessary, but since in practice most programs have some nondeterministic behavior, the algorithm performs the check, in order to not modify the state

machine erroneously. The split is also not performed, if the considered state has only one sequence, since splitting would then create an exact copy of the candidate and leave behind an empty state.

If all conditions for splitting are met, a new state $\tilde{\mathfrak{s}}$ is created. The sequence set $\tilde{\mathcal{S}}$ will contain the sequence s' (Line 4). The new behavior is added to the new state (Line 5). For each old behavior in $\mathfrak{B}$ that was previously observed with s', the algorithm adds a new behavior with only s' as sequence context to the new state (Line 6). Afterwards, the sequence s' is removed from the sequence set of the old state $\mathfrak{s}'$ and the new state $\tilde{\mathfrak{s}}$ is added to the set of states $\mathfrak{S}$ (Lines 8 and 9). Finally, the sequence s' is removed from all behavior contexts of the old state, and if the context would be empty afterwards, the behavior is removed entirely (Line 10).

Otherwise, if the message behavior was seen in the state $\mathfrak{s}'$ before, the algorithm adds the sequence s' to the corresponding observed behavior's sequence set (Line 13). If the message behavior was not seen in the state $\mathfrak{s}'$ before, it is added to the set of observed behaviors (Lines 15 and 16).

In Algorithm 8 and Algorithm 10 the algorithms select the state $\mathfrak{s}'$ by looking for a state that contains the sequence s'. There is always exactly one state that satisfies this condition, i.e. the states are uniquely identified by one or more sequences. In other words, each sequence can reach exactly one state (since our state machine is deterministic) and each state is reachable by one or more sequences, i.e. there exist one or more paths through the state machine for each state. Formally this can be written as

$$\forall \mathfrak{s}_1 = (\mathcal{S}_1, \mathcal{B}_1) \in \mathfrak{S}, \forall \mathfrak{s}_2 = (\mathcal{S}_2, \mathcal{B}_2) \in \mathfrak{S} : \forall s \in \mathcal{S}_1 \cup \mathcal{S}_2 : s \in \mathcal{S}_1 \wedge s \in \mathcal{S}_2 \Rightarrow \mathfrak{s}_1 = \mathfrak{s}_2.$$

The algorithm only modifies states in Algorithm 7, Algorithm 8, Algorithm 9, and Algorithm 10. In Algorithm 7 and Algorithm 9 the sequence is new, meaning it is not contained in any state yet. Adding the sequence to any one state, will not violate the invariant. In Algorithm 8 the sequence was already sent previously by the fuzzer. The algorithm fetches the state with the sequence and only adds a behavior to the corresponding state. This also does not violate the invariant, since the set of sequences is never changed. Finally, in Algorithm 10 the algorithm uses the sequence again to fetch the corresponding state. If the algorithm splits the state (if branch), the sequence is removed from the old state and added to the newly created state. This preserves the invariant. If the state is not split (else branch), the set of sequences is not modified, also preserving the invariant.

4.4 Fuzzer-Target-synchronisation

The algorithm introduced in the previous section depends on whether the message behavior can reliably be extracted. As we will now see, without some additional modifications to the SUT this is not always possible. The best case would be if the SUT blocks until a message is received and only then continues with its execution. However, if we look at the example program (Listing 4.1) again this is not the case, since the `recv` function only blocks for 50 milliseconds. If no message was received, the message processing is skipped, and no response is sent, but the main loop will call the `do_background_processing` function.

Let us consider the sequence ⟨OPN⟩ with message MSG again. In Table 4.1, the coverage for this sequence-message pair is (1,2,2,2,1,1,1,1,0,0,0,0,0,2,2,2). However, it is possible that it takes more than 50 milliseconds for the network to deliver the message. If that is the case, the example program will return `None` from the `recv` call because the timeout was reached. Then the program will iterate once, and only call `do_background_processing`. Afterwards, the program will once again block on `recv`. In case the message arrives now, this will result in a coverage profile of (1,3,3,3,1,1,2,1,1,0,0,1,0,3,2,3).

If we calculate the message behavior as described in Section 4.2 for this scenario, we obtain the message behavior (0,2,2.2,0,0,2,1,1,0,0,1,0,2,1,2,) for the sequence ⟨OPN⟩ with message MSG. However, recall that previously we arrived at the message behavior (0,1,1,1,0,0,1,1,0,0,0,0,0,1,1,1) for the sequence ⟨OPN⟩ with message MSG. These differing message behaviors for the same sequence violate the assumption that the program is deterministic and will reduce the effectiveness of the algorithm described in Section 4.3. Furthermore, the nondeterminism introduced by the timing behavior of the network might reduce the effectiveness of the fuzzer in general.

```
1 signal_main_loop_completed()
2 wait_for_fuzzer_to_allow_iteration()
```

Listing 4.2: Injected synchronisation code.

In order to alleviate this problem, we introduce a synchronisation mechanism between the fuzzer and the SUT. In addition to the coverage instrumentation the fuzzer injects when compiling the target program $\mathcal{P}$ during the Preprocess step (see Section 2.4), the fuzzer will inject the set of instructions listed in Listing 4.2 in the main loop of the target program $\mathcal{P}$. In the case of our example program (Listing 4.1) the code is injected between Line 2 and Line 3. The injected code signals the fuzzer that the program has completed one iteration of the main loop and then waits for the fuzzer to signal that another iteration can be performed.

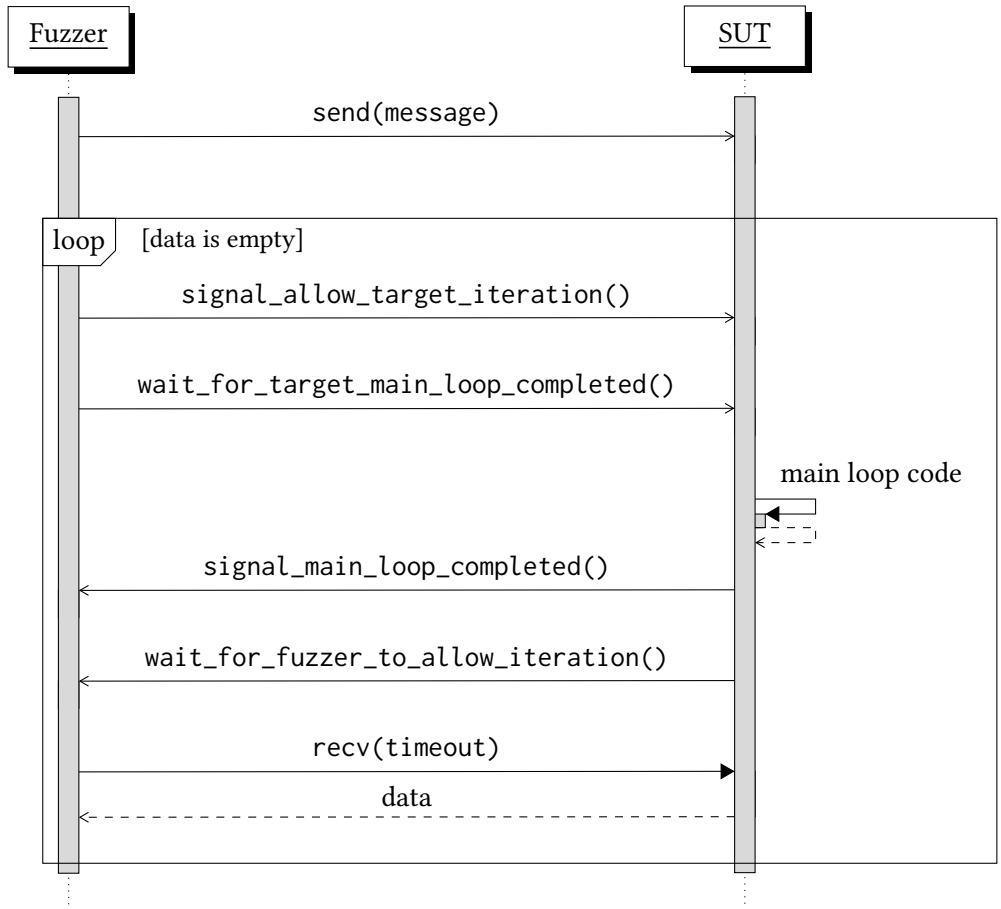


Figure 4.7: Sequence diagram of the interaction between fuzzer and target when sending a single message.

On the fuzzer side the required code is listed in Listing 4.3. We focus on the general concept here and refer to the implementation[2] for error handling and other implementation details. The fuzzer performs some initial setup and waits for the SUT to wait for the fuzzer. Afterwards, the code in Listing 4.3 is executed for each message. First, the fuzzer sends the message, allowing for some time such that the message can arrive at the target. Then the fuzzer signals the SUT that it can perform a single iteration of its mainloop, after which the fuzzer waits for the SUT to complete its iteration. As soon as the SUT signals that it has completed an iteration of its mainloop, the fuzzer tries to receive data from the SUT. This is repeated until the SUT sends a response, after which the fuzzer proceeds with sending the next message in the same way. The interaction between the fuzzer and the SUT is depicted in Figure 4.7.

2 https://github.com/mlgiraud/AFLplusplus-stateful

```
send(message)
data = None
while not data:
    signal_allow_target_iteration()
    wait_for_target_main_loop_completed()
    data = recv(timeout)
```

Listing 4.3: Fuzzer side of the synchronisation mechanism for a single message.

4.5 Mutation Strategy

Since one of the assumptions required for the state identification algorithm described in Section 4.3 is that sequences are only modified at the end, we modify the mutation strategy a bit in contrast to AFLNet. We leave the mutation strategies for the content of messages the same, i.e. the same as AFL and AFLNet.

For the sequence mutation, AFLNet uses the operations insert after, insert before, replace, and duplicate. We omit the duplicate operation for our approach, since this is covered by insert after and insert before when the selected message is the same as the currently mutated one. Also, instead of separating the seed into three parts (see Section 3.2), we always mutate the last message of the seed. This also allows for a simpler implementation, and the optimization discussed in section Section 4.6.

Instead of adding the sequence mutations to the set of normal mutations that is uniformly sampled, we instead use a scoring function to determine when to do sequence mutation and when to use the normal afl mutations. This means that we, in contrast to AFLNet, preserve the probabilities for the original mutations when performing havoc mutations. However, similar to AFLNet the information gained by building a state machine is incorporated into the mutations by shifting the threshold for doing sequence mutations. The probability to do sequence mutation is calculated as

$$P(\text{mutate sequence}) = \max\left(\frac{1}{1+e^{-0.5(B-4S-\log_{10}(a+1))}}, 0.1\right),$$

where B is the number of behaviors in the currently selected state, S is the number of sequences in the currently selected state, and a is the number of times the state was fuzzed. The probability to use the normal AFL havoc mutations is then $P(\text{mutate havoc}) = 1 - P(\text{mutate sequence})$. The chosen function is a logistic function with $4S$ as the midpoint and a growth rate of 0.5. The midpoint is shifted in favor of havoc mutations proportional to the number of times the

state was fuzzed, i.e. the older the state gets the more havoc mutations are prioritized, by the logarithmic term.

The intuition for choosing this function is that a high ratio of sequences to behaviors indicates that we have to first find new behaviors in order to try and differentiate between them. On the other hand, if the ratio of sequences to behaviors is low, i.e. if we have more behaviors than sequences, it makes sense to try to find a sequence that produces a different behavior for a message that was sent in the current state. Also, instead of choosing a hard threshold where only either sequences mutations or havoc mutations are performed, the logistic function allows for a soft transition around the midpoint. This means that there is always a slight possibility to still do sequence mutations when the threshold is in favor of havoc mutations and vice versa. The sequence probability is capped at a minimum of 0.1 in order to always leave the possibility of sequence mutations. It should be noted that the values in the function as well as the shifting of the midpoint were arrived at by intuition and testing out a few combinations. The focus of this thesis is primarily on the state identification part. Due to time constraints these values were not investigated further.

4.6 State Forkserver

Chen et al. describe a different approach to stateful greybox fuzzing [ClV19]. Their approach uses a multi-state forkserver, that can fork the SUT at specific states in order to improve fuzzing performance. We try to adapt a similar approach, where the SUT is set to a specific state and then forked from that state.

For a sequence $s = (m_1, \ldots, m_n)$ and message m the fuzzer first sends the messages $m1, \ldots, m_n$. It then signals the SUT to enter forkserver mode. Afterwards, the forkserver can be used to spawn new process instances of the SUT in this state. The actual fuzzing is then performed on the message m. Each fuzzing iteration thus only has to clone the existing process and send the fuzzed message, instead of having to send the complete sequence s each time.

5 Implementation

In Chapter 4 we propose an approach to automate the state identification component of AFLNet. We also propose additional changes required in order to make the proposed approach work. In order to evaluate the concepts in Chapter 6, we implement them on AFLPlusPlus, in order to then compare them with AFLNet. This chapter discusses why we choose AFLPlusPlus instead of modifying AFLNet in Section 5.1. Furthermore, we give an overview of the modifications done to AFLPlusPlus, and present the datastructures used for implementing the state machine in Section 5.2. The actual implementation is available on github.[1]

5.1 AFLPlusPlus vs. AFLNet

We now briefly discuss why AFLPlusPlus was chosen as implementation basis instead of AFLNet. AFLNet is based on AFL, a predecessor of AFLPlusPlus. AFLPlusPlus comes with many addons to AFL that implement recent research [Heu+20]. Additionally, compared to AFL, AFLPlusPlus is built in a more modular fashion, making it easier to implement modifications and new functionality. AFLPlusPlus seems to be the superior fuzzer, since it implements all the features of AFL and more. We chose not to modify AFLNet, since the state machine works differently to our approach. This would have necessitated changing most of the code added by AFLNet. Instead, we chose to port the functionality added by AFLNet relevant to sending messages over the network to AFLPlusPlus, and add the state machine implementation on top of that. This also makes it possible to later integrate the developed stateful fuzzer as an AFLPlusPlus addon in order to increase adaptation.

5.2 AFLPlusPlus modifications

This section briefly highlight where changes were made to AFLPlusPlus. We also highlight the structure of the state machine. The actual implementation is not discussed, since it does not differ conceptually from the pseudo code introduced in Chapter 4.

1 Private repository: `https://github.com/mlgiraud/AFLplusplus-stateful`. For access please contact mark.giraud@iosb.fraunhofer.de

The entry point of AFLPlusPlus is contained in `afl-fuzz.c`. This file contains the code that parses command line parameters and the main loop of the fuzzer. The parameter parsing needed for AFLNet relevant parts was integrated into this file. Additionally, a separate main loop was introduced that runs when fuzzing stateful software, instead of the default fuzzing loop. The main difference is that after each queue cycle the selected state is chosen, according to AFLNet's scoring policy.

The actual fuzzing happens in the `fuzz_one` function in `afl-fuzz-one.c`. Instead of using the seed selection from AFLNet, the same strategy was implemented differently, by using the already existing functionality of seed skipping. At the start of the `fuzz_one` function, a function is called, that performs the same checks AFLNet would do, and then skips the seed, if it did not pass. This also effectively divides the seed queue up into a queue for each state. As we will shortly see, each queue entry now has a state associated with it, similar to AFLNet, such that any seed not belonging to the currently selected state is simply ignored. In addition, the region extraction was implemented similarly to AFLNet, with the key difference that now only the last region is mutated as motivated in Chapter 4. Lastly, the havoc mutation phase was adapted to include the sequence mutations as described in Section 4.5.

In order to enable fuzzing over the network, most of the code from AFLNet was adapted into `afl-forkserver.c`. This file contains most of the code responsible for communicating with the SUT and for spawning new instances of the SUT. The function responsible for sending individual messages was modified in order to accommodate for the synchronisation mechanism introduced in Section 4.4. As synchronisation primitive, two semaphores were used. One of the semaphores is used by the fuzzer to signal the SUT that it may loop, i.e. the fuzzer posts the semaphore, and the SUT waits on the semaphore. The other semaphore models the other direction, i.e. the fuzzer waits on the semaphore for the SUT to post, corresponding to the fuzzer waiting for the SUT to pause. The SUT part of the synchronisation code was added to `afl-llvm-rt.o.c`, which gets compiled into the `afl-clang` compiler used to compile the SUTs. The compilation process then injects the actual code into the SUT.

The implementation of the state machine is split in two files. First, the `afl-fuzz.h` file was modified to include all necessary declarations. The observed behaviors $\mathfrak{b}$ (see Definition 4.3) were implemented as a struct containing a hashset of sequence hashes, the message in a hashed form, and the behavior in a hashed form. States were modeled as a struct containing a hashmap of sequences indexed by the sequence hash (corresponding to $\mathcal{S}$ in Definition 4.3), a hashmap of behaviors that is indexed by using the message hash (corresponding to $\mathcal{B}$ in Definition 4.3), multiple statistics values, and a few helper variables. The sequence hash map maps a hash of a sequence to another map. This map maps message hashes to the corresponding queue entries. This way it is easy to retrieve queue entries via a combination of sequence and message, and

when splitting the entries can easily be rearranged. The actual state machine is modeled as another struct that contains a list of states, and two hashmaps corresponding to the helper sets σ and μ in Algorithm 5. Instead of implementing σ and μ as sets, they were implemented as hashmaps in order to easily retrieve the state associated with the sequence or message.

The implementation of Algorithms 5 to 10 is contained in the new file `afl-fuzz-net.c`. In addition, the file also contains the region extraction functions from AFLNet, which had to be slightly adapted to accommodate for the different structures in AFLPlusPlus.

Finally, the state snapshotting mechanism was prototypically implemented on top of the aforementioned mutations by modifying the function injected into the main loop of the SUT. In essence, the function that is called in each main loop iteration is modified to include a check if a flag is set. If this flag is set, the SUT will enter a forkserver function similar to the one already present for the unmodified AFLPlusPlus. Instead of using the default forkserver to fork a new SUT instance, this new forkserver is used. The flag that determines when to enter the forkserver mode is set after sending the last input of the sequence just before sending the fuzzed message.

5.3 AFLNet modifications

Some minor modifications had to be made in order to run the evaluation. First, during initial evaluations AFLNet kept crashing randomly after a few hours, making it impossible to run 48 hour evaluation runs. This issue[2] was caused by errors when writing the test file to disk. However, since this functionality is only required for SUTs that read their input from a file, it is not required for evaluating our targets. The solution was to simply remove the code in question.[3] Second, since AFLNet currently does not save timestamps for seed files, they had to be added to the output in order to later plot the coverage achieved over time.[4] Lastly, the OPC UA protocol is currently not supported by AFLNet yet. In order to add support for OPC UA, later needed for our evaluation in Chapter 6, the *request sequence parser* and *state machine learning* components had to be extended. The *request sequence parser* was extended by adding a new `extract_requests_opcua`[5] function that splits an input into separate OPC UA requests. In order to do this the `UA_TcpMessageHeader_decodeBinary` function, which is part of the

2 https://github.com/aflnet/aflnet/issues/20
3 https://github.com/mlgiraud/aflnet-1/blob/863bbb2b77da8d4a106224221aa0273fefaf5f1c/afl-fuzz.c#L3335
4 https://github.com/mlgiraud/aflnet-1/blob/863bbb2b77da8d4a106224221aa0273fefaf5f1c/afl-fuzz.c#L3980
5 https://github.com/mlgiraud/aflnet-1/blob/863bbb2b77da8d4a106224221aa0273fefaf5f1c/aflnet.c#L582

open62541 parser, was used. This function extracts the message type, and the message size of a message. The extracted message size is then used to extract the rest of the message. Very similarly, the *state machine learning* component was modified. In order to support OPC UA, the `extract_response_codes_opcua` function had to be implemented. The function uses the same decoding function of the open62541 parser as before and extracts the individual responses. The message type is now used as the response code that the state machine uses to identify states. Except for some additional boilerplate code needed to integrate the modifications, no additional changes were made to AFLNet.

6 Evaluation

This chapter evaluates the previously developed approach on three SUTs. The first SUT open62541 is an open source implementation of the OPC UA protocol stack [Iat+20]. The second SUT live555 is an open source implementation of the commonly used streaming protocol RTSP [Pro20]. The third and last SUT lightftp is an open source implementation of an FTP server [Gau20]. The latter two SUTs are both evaluated by the AFLNet paper [PBR20b] as well, which enables a direct comparison of our newly developed approach against AFLNet. For the evaluation we follow most of the guidelines presented for fuzzing evaluation by Klees et al. [Kle+18].

We first discuss the evaluation method in Section 6.1. Afterwards, we present the achieved results in Section 6.2. Finally, we analyse the individual crashes and hangs discovered by the fuzzers, checking for false positives in Section 6.3.

6.1 Method

In order to answer the research questions posed in Section 3.3, we perform multiple evaluation runs. The interpretation and discussion of the results listed in this chapter follow in Chapter 7. The implementation of the methods introduced in Chapter 4 will be called SNAPP (Stateful-Network-AFLPlusPlus) for the remainder of the thesis, and the forkserver version will be called FSNAPP. For convenience, we list the research questions once more:

RQ1 Can the state identification process be automated by using already provided coverage information?

RQ2 How effective is a fuzzer using an automated state identification process compared to the manual approach of AFLNet if measuring performance by using coverage achieved as metric?

RQ3 Does the seed corpus selection make a difference in achieved coverage when fuzzing stateful software?

RQ4 Could program state snapshotting improve fuzzing speed and as such effectiveness?

In order to answer RQ1 and RQ2, we perform at least one evaluation run for SNAPP and AFLNet respectively for each of the three SUTs open62541, lightftp, and live555. Table 6.2 lists all the SUT-corpus-fuzzer configurations used for the evaluation. The inputs generated by each fuzzer during the evaluation runs are then executed on a coverage instrumented executable in order to compare the coverage achieved by each fuzzer. We chose lightftp and live555 as SUTs, because they are the primary SUTs used for the Evaluation of the AFLNet implementation [PBR20b]. By comparing AFLNet and SNAPP on these SUTs, we can draw indirect comparisons with the fuzzers, boofuzz and AFLNwe, which AFLNet was evaluated against. Since AFLNet is currently state of the art for stateful greybox fuzzing, we use it as baseline to compare our approach against.

We additionally include the open62541 SUT, because it is a single threaded implementation of the highly stateful OPC UA protocol (see Section 2.2). For further evaluations, we only use this SUT, since it is the most stateful of the three and has the largest codebase (see Table 6.1). It also behaves in the most deterministic way, since it uses no multithreading and supports disabling randomization out of the box, making it an ideal fuzzing target.

Klees et al. recommend using the same version of SUTs in order to make informal or indirect comparison possible [Kle+18]. For lightftp and live555 we thus use the same commit as used by Pham et al. [PBR20b]. For open62541 we use the most recent version as of August 19, 2020. All SUTs are listed in Table 6.1 with the corresponding version used for fuzzing. Klees et al. also argue, that a larger set of SUTs should be used. However, we only use the three SUTs in Table 6.1, arguing that due to their stateful nature they are a suitable sample for the software we want to test. Also, evaluating more SUTs would not have fit into the timeframe of this thesis.

SUT Name	Protocol	LOC	BBS	Commit Hash
open62541	OPC UA	54 904	30 255	`ee275e79310280a71fad3b21f6d430c0c046091b`
lightftp	FTP	2 236	1 147	`5980ea1a0ee0e5c3015275f93445626f8c25c83a`
live555	RTSP	31 119	15 903	`ceeb4f462709695b145852de309d8cd25e2dca01`

Table 6.1: Evaluation SUTs and their protocols, lines of code (LOC), number of basic blocks (BBS) and the git commit hash of the version used for the evaluation. All protocols are based on TCP.

Since Klees et al. [Kle+18] recommend running a fuzzer for at least 24 hours we run each fuzzer for 48 hours, because the network enabled fuzzers are slower than regular afl due to networking delays. We also run a longer evaluation on the open62541 SUT, in order to check if

and how much the performance varies after 48 hours. This longer run is only be performed on open62541 due to time constraints.

For the seed corpora we manually select messages and message sequences specific to each SUT. Using an empty seed corpus would most likely not yield meaningful results for measuring the effectiveness of a stateful fuzzer [Kle+18; Myt+09], since only correct message sequences will be processed in a meaningful way by the SUT. The evaluation runs for lightftp and live555 are performed on the same seeds used by Pham et al. [PBR20b], indicated by the "aflnet" entries in the corpus column in Table 6.2. We slightly modify the corpus by adding the individual messages contained in the corpus' seed sequences as seeds. The reasoning behind this is that SNAPP needs individual messages such that the state identification algorithm can be initialized properly. In future work this could be done automatically by SNAPP, but was not implemented due to time constraints.

SUT	Corpus	Fuzzer
open62541	7d-large	SNAPP AFLNet
	large	SNAPP AFLNet
	small	SNAPP fsnapp AFLNet
lightftp	aflnet	SNAPP AFLNet
live555	aflnet	SNAPP AFLNet

Table 6.2: SUT-corpus-fuzzer configurations used to produce the evaluation results in Section 6.2.

In order to answer RQ2 we plot the covered basic blocks over time. Each comparison figure consists of two plots. The first one plots the best, worst and mean run. The second one plots the mean and confidence interval. This way we can argue about outliers (runs that perform much better or worse compared to the mean), and most interestingly the mean performance of AFLNet compared to SNAPP.

To answer RQ3 we perform two evaluation runs on open62541 with different corpora. The first corpus listed in Table 6.3 only contains messages required to establish a *SecureChannel* and a *Session*, since these are required for almost all services [Fou17b]. Additionally, it contains a few

Message Type	Description
HEL message	Required before OpenSecureChannel.
OpenSecureChannel	Required before any other Message except HEL.
CloseSecureChannel	Closes the Channel and TCP connection.
CreateSession	Required before ActivateSession.
ActivateSession	Required for most services.
CloseSession	Closes an established session. The channel stays open.
GetEndpoints	Lists the endpoints of the server. Does not require a session.
Read	Used to read attributes of a node in the servers nodeset.
Write	Used to write attributes of a node in the servers nodeset.
HistoryRead HistoryUpdate	Same as Read and Write but for historical values
Publish	Used to get notification on existing subscriptions.
CreateMonitoredItems	Creates and adds a monitored item to a subscription.
DeleteMonitoredItems	Removes a previously created monitored item.

Table 6.3: Message types contained in the small fuzzing corpus.

basic service requests. The *Publish*, *CreateMonitoredItmes*, and *DeleteMonitoredItems* requests require a subscription to exist. The messages to create such a subscription were intentionally omitted in order to analyze how the fuzzer behaves in cases where an intermediate message is not contained in the initial corpus.

The second corpus listed in Table 6.4 contains more service requests in addition to those listed in Table 6.3. The remaining request types of the discovery service were included. In addition, the complete view service set was added. The view service is used to navigate through the address space of the SUT. The address space can be modified by the attribute service, which is already contained in Table 6.3. However, the NodeManagement service is also able to modify the address space, which is why we include it here. Further, since the MonitoredItem requests in the small corpus Table 6.3 require a subscription, we add all request types of the subscription set to the corpus, and the remaining MonitoredItem requests. With this bigger corpus the fuzzer has a template for every request type (see Section 2.2) that the SUT can handle.

Finally, in order to answer RQ4, we perform another evaluation run on open62541 with the small corpus (Table 6.3). However, this time instead of using SNAPP, we run a slightly modified version called FSNAPP that implements the state snapshotting technique described in Section 4.6.

Message Type	Description
FindServers FindServersOnNetwork RegisterServer	Discovery service requests
Browse BrowseNext RegisterNodes UnregisterNodes TranslateBrowsePathsToNodeIds	View service requests
CreateSubscription ModifySubscription SetPublishingMode Republish DeleteSubscriptions	Subscription service requests.
ModifyMonitoredItems	Modifies monitored items. Create/Delete in Table 6.3
SetMonitoringMode	Sets the monitoring mode of monitored items.
Call	Part of the MethodService. Calls a remote procedure.
AddNodes AddReferences DeleteReferences DeleteNodes	NodeManagement service requests.

Table 6.4: Message types contained in the large fuzzing corpus in addition to those in Table 6.3.

6.1.1 SUT Adjustments

In order to make the SUTs fuzzable, some minor adjustments to the SUTs are required. We discuss the necessary changes on a high level. The implementation details are contained in the artifacts accompanying this thesis.

open62541

For open62541 there are already some adjustments available since it is continually fuzzed by oss-fuzz [Aiz+16]. These adjustments can be enabled by supplying a build flag, which are enabled for fuzzing and coverage runs. An additional flag that disables random generator seeding is also enabled. This ensures that the program is as deterministic as possible.

For the session service this flag enables saving the authentication token and reusing it later on such that the fuzzer does not have to guess the token. Usually the token received after

sending a *CreateSessionRequest* has to be used in the following *ActivateSessionRequest* in order to proceed [Fou17b]. This check is essentially disabled by internally overwriting the token in the received request with the saved token that was returned in the previous response. For the *SecureChannel* service the flag disables the sequence number and channel id checks of the protocol. These checks are normally performed for security reasons [Fou17c]. They however introduce a value dependency between messages that is hard for fuzzers to overcome.

In addition to the aforementioned changes, we disable the *SecureChannelToken* verification and introduce a deterministic clock. The *SecureChannelToken* is removed, because for each message of type MSG, the token id in the request header is matched with the currently active token id in the server [Fou17c]. We simply remove this check such that the fuzzer does not have to guess the exact token id in order to make progress, since guessing the correct id is very unlikely. The clock is also replaced by a simple counter that is increased each mainloop iteration. Otherwise the SUT would exercise nondeterministic behavior every time the clock is sampled.

Finally, for SNAPP the mainloop injection function is placed inside the `ua_server.c` file inside the main server loop. Also, the wait time for select is disabled in order to speed up fuzzing. This could only be done for SNAPP. For AFLNet, disabling the select timeout resulted in decreased stability and slower execution times.

The OPC UA protocol also supports encryption of messages [Fou17c]. We do not enable this for our fuzzing runs and only concentrate on the unencrypted communication. Due to the nature of cryptography, guessing a correctly encrypted message is very unlikely, hindering the fuzzer from making any kind of meaningful progress. Disabling these checks means that fuzzing will not be able to reach and execute code that is related to the encryption and decryption of messages. In order to test this code, a different approach would be required. However, omitting this code should not have much effect on the results for any code that is executed after these security checks, i.e. the number of basic blocks and bugs found after the security mechanisms should not vary much.

lightftp

For lightftp we employ the same changes the AFLNet authors used in order to evaluate their fuzzer. We slightly modify them, in order to provide a cleaner shutdown for the SUT when terminated by the fuzzer. The SUT spawns a thread for each connection. Since we do not want multiple connections in order to be as deterministic as possible, the code is modified such that every time it spawns a new thread, the spawning thread waits for the spawned thread to finish, and then exits itself. This means that only one thread will be active at a time, and a

thread can only ever spawn one other thread, meaning the program is essentially executed sequentially. We also make sure that the SUT cleanly exits when terminated by the fuzzer, by adding handlers for the `SIGINT` and `SIGTERM` signals. Finally, for SNAPP the `recvcmd` function is modified. The blocking `recv` call is replaced by a non blocking call that is called in a loop. The loop exits, once a message was received or the socket was closed. In each iteration, before calling recv, the mainloop injection function is called in order to synchronise the SUT with the fuzzer.

live555

For live555 we also employ the same changes the ALFNet authors used in order to evaluate their fuzzer. The only change required for both fuzzers is fixing the session id to the value 8888. For SNAPP the mainloop injection function is placed inside the while loop in the `BasicTaskScheduler0::doEventLoop` function. Furthermore, the timeout for the select call in the `BasicTaskScheduler::SingleStep` function is set to zero in order to speed up the fuzzing process. Finally, a handler for the `SIGINT` and `SIGTERM` signal is added, that sets a stop flag, such that the server cleanly shuts down when signaled to do so by the fuzzer.

6.1.2 Evaluation Environment

The different fuzzer-SUT-corpus configurations are evaluated on the system specified in Table 6.5. During evaluation no other cpu intensive tasks were performed on the system.

Resource Type	Description
Kernel	Linux `4.15.0-122-generic`
Distribution	Ubuntu 18.04.5 LTS
CPU	Intel Xeon CPU E5-2630 v4 (2464 MHz, 40 cores)
Ram	264 032 996 kB

Table 6.5: Evaluation system specification.

As previously discussed, each fuzzer-SUT-corpus combination is run for at least 48 hours on 20 different instances. Since there is a total of 11 of such combinations (see Table 6.6), running everything sequentially is not feasible. Doing so would take at least $48 \text{ hours} \cdot 20 \cdot 11 = 440$ days. Speeding up the evaluation process by simply running everything in parallel is also not possible without further adjustments. Because the SUTs require communication to happen over the network, running multiple instances in parallel on the same machine requires the ports of one fuzzing instance to not conflict with those of another.

In order to alleviate this, we run each fuzzing instance in a separate docker container. The docker container isolates the hosts resources such that each instance can operate as if running on a dedicated machine. Each container is bound to one dedicated cpu core, such that the effect of kernel scheduling is minimized. To achieve a fair comparison, each fuzzing combination is run in isolation, i.e. only 20 instances are ever run in parallel, and they all belong to the same fuzzer-SUT-corpus configuration.

By running the evaluation in parallel using docker as a virtualization layer, the evaluation time is reduced to roughly 48 hours $\cdot$ 9 + 7 days $\cdot$ 2 = 32 days. The running time for two out of the 11 fuzzer-SUT-corpus configurations was chosen to be one week each in order to analyze the long term behavior of the fuzzers.

6.2 Results

This section lists the results achieved by the evaluation runs. The coverage used in the following tables and figures was collected by supplying the inputs discovered by the fuzzer to a llvm-cov instrumented executable. This executable has all modifications for fuzzing enabled, but is not instrumented by the fuzzer. By doing this, each fuzzing run can be executed on the exact same executable. The coverage achieved by each individual input is then accumulated for each following input, and finally plotted as seen in the following figures.

6.2.1 Average Results per Run

Table 6.6 contains the final results achieved for each individual fuzzer-SUT-corpus configuration averaged over all 20 runs. The second column contains the corpora used for the runs. The 7d-large corpus entry in Table 6.6 refers to the corpus described in Table 6.4 and, in contrast to all other corpora, it was executed for seven days. The large and small corpora refer to Table 6.4 and Table 6.3 respectively. For lightftp and live555 we use the same corpus “aflnet” as the original paper by Pham et al. [PBR20b]. The table lists the average hangs, crashes and executions achieved by each fuzzer. A crash refers to an actual crash of the SUT, i.e. the fuzzer detected that the program terminated unexpectedly while sending the inputs. A hang refers to a timeout of the SUT during fuzzing. For the performed experiments this timeout was configured to 1 second. If after 1 second the SUT has not exited yet, the fuzzer will kill the SUT and register the input as a hang. In addition, it also contains the average *stability*, the average total coverage achieved in basic blocks and relative to the total number of basic blocks. *Stability* is a value calculated by AFL that represents how deterministic the program behaves during fuzzing, with 1 meaning the program is completely deterministic, and 0 meaning the

program is completely nondeterministic. The value is determined with the help of the coverage feedback received during fuzzing [Zal+20].

Table 6.6 ist structured in a way such that the runs that are later compared against each other are adjacent. Each of the pairs of SNAPP and AFLNet also has an entry in Table 6.7, where the coverages are compared, and the statistical significance is presented. The only exception is the open62541 run with a small corpus. For these three runs, each combination of fuzzers is compared, as such resulting in three entries in Table 6.7.

The crashes and hangs listed in Table 6.6 are analysed in Section 6.3. In all cases AFLNet is able to perform more executions of the SUT than SNAPP, meaning that AFLNet is able to execute the tests quicker. The stability for almost all open62541 test cases is greater than 0.9, meaning that the program behaves mostly deterministic. However, the AFLNet run over seven days on open62541 has slightly worse stability with a value of 0.86. Also, the FSNAPP run with the small corpus on open62541 has significantly worse stability amounting to only 0.47.

The BB and BB% columns show how much of the SUTs' code was covered. The percentages correspond to the ratio of BBs hit relative to the total amount of BBs of the target (listed in Table 6.1). The values in the BB column are used for the comparisons in Table 6.7.

SUT	Corpus	Fuzzer	Hangs	Crashes	Executions	Stab.	BB	BB%
open62541	7d-large	SNAPP	0.00	4.95	2 830 749	0.97	10 191	34
		AFLNet	166.15	0.00	4 439 589	0.86	9 873	33
	large	SNAPP	0.00	1.10	892 540	0.97	9 979	33
		AFLNet	43.35	0.00	1 621 957	0.91	9 554	32
	small	SNAPP	0.00	0.25	1 184 972	0.97	8 116	27
		FSNAPP	0.11	0.00	3 316 521	0.47	8 547	28
		AFLNet	20.10	0.00	2 306 884	0.96	8 459	28
lightftp	aflnet	SNAPP	0.00	0.00	743 943	0.78	629	55
		AFLNet	60.50	0.00	1 007 982	0.10	627	55
live555	aflnet	SNAPP	117.25	0.00	1 519 658	0.30	3 560	22
		AFLNet	0.00	262.70	2 095 367	0.03	3 521	22

Table 6.6: Final statistics (hangs, crashes, executions, stability, basic blocks, and basic blocks in percent of the total basic blocks) for each SUT-corpus-fuzzer evaluation configuration averaged over all 20 runs.

		Basic Block Coverage		
		% Incr.	p-value	Figure
SNAPP vs AFLNet	open62541	−4.05	0.038	6.1
	lightftp	0.31	0.946	6.2
	live555	1.11	0.091	6.3
FSNAPP vs AFLNet	open62541	1.04	0.222	6.4
FSNAPP vs SNAPP	open62541	5.31	0.005	6.5
SNAPP vs AFLNet (large)	open62541	4.44	< 0.001	6.6
SNAPP vs AFLNet (long)	open62541	3.22	< 0.001	6.7

Table 6.7: Mean coverage increase and statistical significance (p-value) when comparing SNAPP and FSNAPP to AFLNet.

6.2.2 Fuzzer Comparison

Table 6.7 depicts the percent increase of basic block coverage and the statistical significance of the results, when comparing different fuzzers. The first three rows compare SNAPP and AFLNet on the three different SUTs open62541, lightftp, and live555. The next two rows compare FSNAPP (SNAPP with the state snapshotting enabled) against AFLNet and SNAPP respectively on open62541. The last two rows compare SNAPP and AFLNet on the large corpus on open62541. However, the comparison in the last row (SNAPP long vs AFLNet) uses the evaluation data on the large corpus where both fuzzers ran for seven days. The statistical significance in Table 6.7 is computed by means of the Mann-Whitney U test [MW47]. The test is performed on the final 20 coverage samples of each of the two fuzzing runs The coverage data used to calculate the percentages is the average of the final coverage from all 20 samples. For each comparison in Table 6.7 there is a figure comparing the two runs. The first plot in each figure compares the mean, and the best and worst runs of the two fuzzers. The second plot compares the averages and confidence intervals of the two fuzzers. In all cases AFLNet is colored blue, while SNAPP is colored orange.

SNAPP vs AFLNet In Figure 6.1 we can see the comparison of SNAPP and AFLNet on open62541 with the small corpus. This is the only run that resulted in less total achieved coverage for SNAPP compared to AFLNet, corresponding to a −4.05% coverage drop, as seen in Table 6.7 in the first row. The comparison of SNAPP and AFLNet on the small corpus for open62541 is statistically significant, since the p-value 0.038 is less than 0.05. In Figure 6.1b we can see that the mean coverage of SNAPP is always less than that of AFLNet. The confidence

intervals however start overlapping after 24 hours. However, even though SNAPP performs worse on average than AFLNet, in Figure 6.1a we can see that the best run of SNAPP performed better than the best run of AFLNet.

In Figure 6.2 we can see the comparison of SNAPP and AFLNet on lightftp. The run resulted in 0.31% higher coverage for SNAPP when compared to AFLNet, with the corresponding p-value being 0.946, as seen in Table 6.7 in the second row. As we can see in Figure 6.2b, SNAPP performs worse than AFLNet for the majority of the test time on average. However, after 40 hours SNAPP catches up to AFLNet and even achieves slightly better coverage on average. The best run for SNAPP and AFLNet as seen in Figure 6.2a achieve similar coverage, but when considering the worst runs, SNAPP performs better than AFLNet.

Finally, the comparison of SNAPP and AFLNet on live555 is depicted in Figure 6.3. The run resulted in 1.11% higher coverage for SNAPP when compared to AFLNet, with the corresponding p-value being 0.091, as seen in Table 6.7 in the third row. SNAPP again on average performs worse in the beginning, but catches up to AFLNet after 10 hours as seen in Figure 6.3b. The worst run of SNAPP performs better than the worst run of AFLNet as seen in Figure 6.3a. The best run for SNAPP also performs better than the best run of AFLNet, but the difference is not as noticeable.

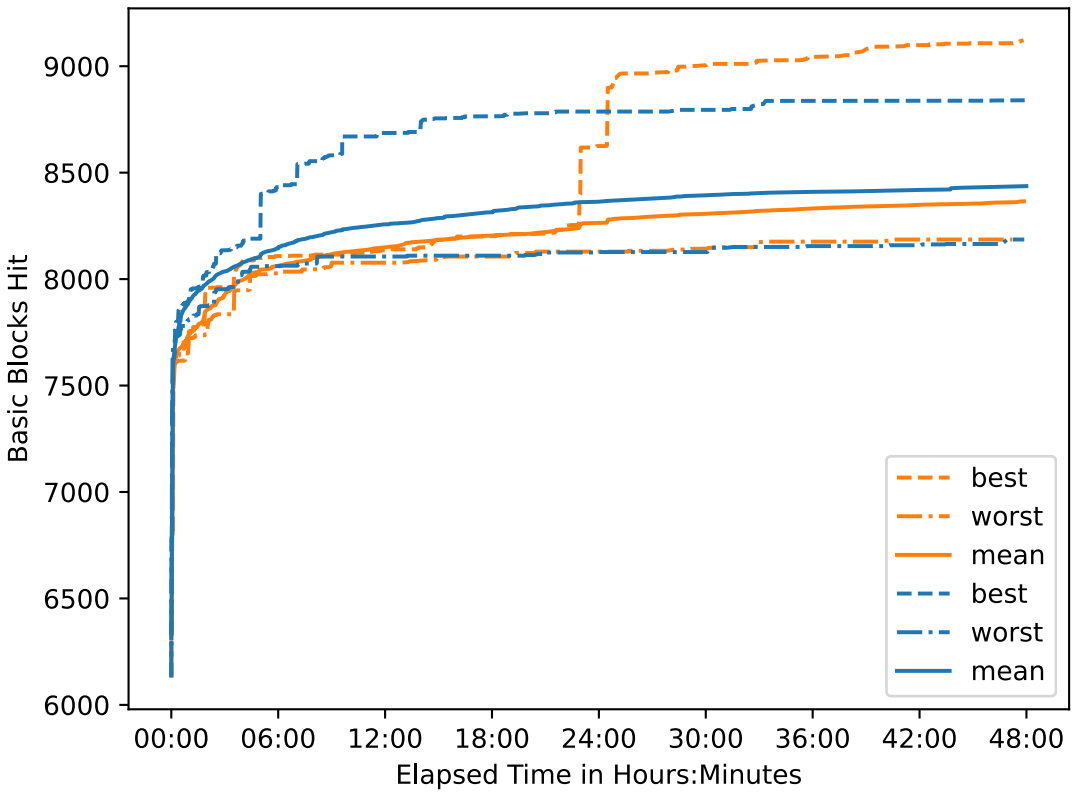
FSNAPP vs SNAPP and AFLNet Figure 6.4 depicts the comparison of FSNAPP with AFLNet with the small corpus on open62541. The run resulted in 1.04% higher coverage for SNAPP compared to AFLNet, with the corresponding p-value being 0.222, as seen in Table 6.7 in the fourth row. As in the previous cases, FSNAPP performs worse than AFLNet in the beginning, but in contrast to SNAPP (see Figure 6.1), FSNAPP catches up to AFLNet after 35 hours as seen in Figure 6.4b. After 35 hours, the coverage for the best run of FSNAPP jumps a large amount, and after 44 hours once more, setting it above the best run of AFLNet as seen in Figure 6.4a. The worst run for both FSNAPP and AFLNet are similar, with the FSNAPP run performing slightly better.

In Figure 6.5 SNAPP and FSNAPP are compared with the small corpus on open62541. The run resulted in 5.31% higher coverage for FSNAPP compared to SNAPP, with the corresponding p-value being 0.005, indicating that the result is statistically significant, as seen in Table 6.7 in the fifth row. On average FSNAPP outperforms SNAPP as seen in Figure 6.5b. However, the best run of SNAPP outperforms the best run of FSNAPP after 24 hours as seen in fig. 6.5a. The worst run of FSNAPP and SNAPP perform similarly, with the FSNAPP run being slightly better.

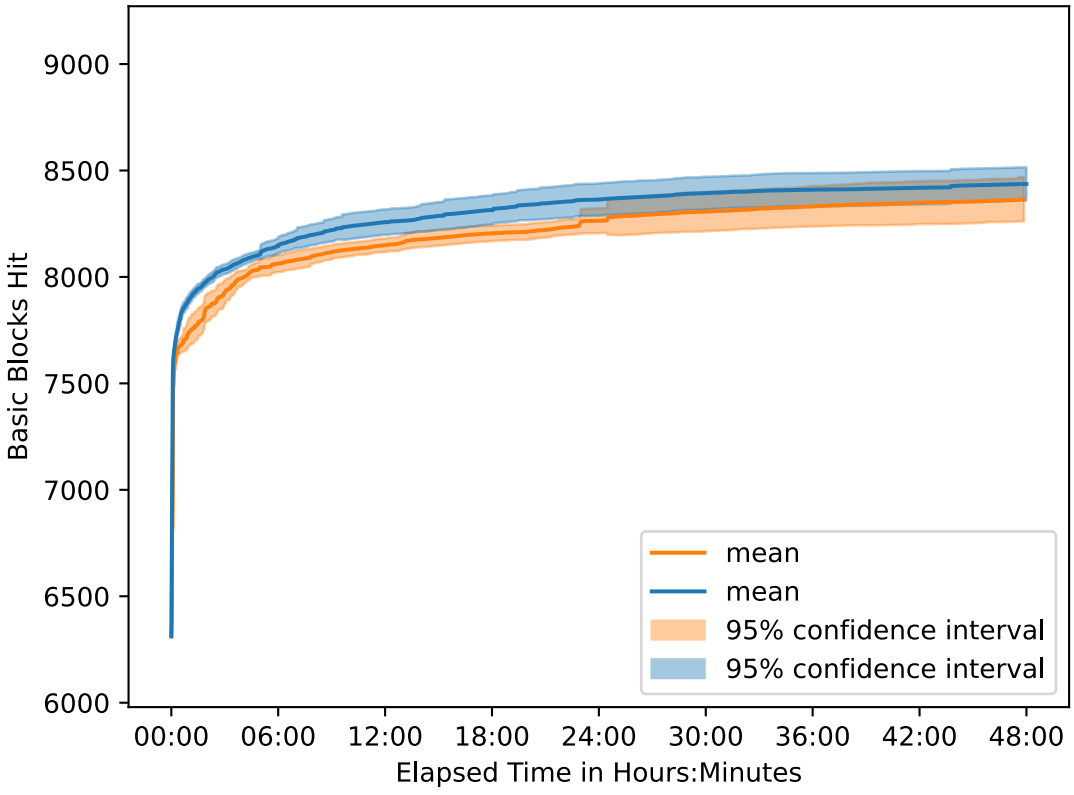
SNAPP vs AFLNet (large corpus) In Figure 6.6 the comparison of SNAPP with AFLNet with the large corpus on open62541 is depicted. The run resulted in 4.44% higher coverage for SNAPP compared to AFLNet, with the corresponding p-value being less than 0.001, indicating that the result is statistically significant, as seen in Table 6.7 in the fifth row. On average, SNAPP outperforms AFLNet as seen in Figure 6.6b. The confidence intervals also do not overlap in contrast to the previously discussed runs. The worst run of SNAPP performs better than the average of AFLNet, and the best run of AFLNet performs worse than the average of SNAPP as seen in Figure 6.6a.

This evaluation run was repeated in the same configuration, but for a longer duration. The results can be seen in Figure 6.7. The run resulted in 3.22% higher coverage for SNAPP compared to AFLNet, with the corresponding p-value being less than 0.001, indicating that the result is statistically significant, as senn in Table 6.7 in the sixth row. Similar to the short run, the achieved coverage for SNAPP on average is higher than for AFLNet, and the confidence intervals do not overlap as seen in fig. 6.7b. However, the worst run of SNAPP in this case is now slightly worse than the average of AFLNet, but still better than the worst run of AFLNet as seen in Figure 6.7a. The best run of SNAPP is still better than the best run of AFLNet, but compared to the short run, the best run of AFLNet now performs better than the average of SNAPP.

For the long-running test we also plot the individual coverages achieved per file in percent of total basic blocks in the file. The coverage is displayed as box-plot in Figure 6.8. For each file the coverage is plotted per fuzzer. The upper bound of a box corresponds to the upper quartile, whereas the lower bound of the box corresponds to the lower quartile. The line inside the box corresponds to the median. The whiskers' length is determined by the last value that lies inside 1.5 times the interquartile range. All other data points outside this range are depicted as points.

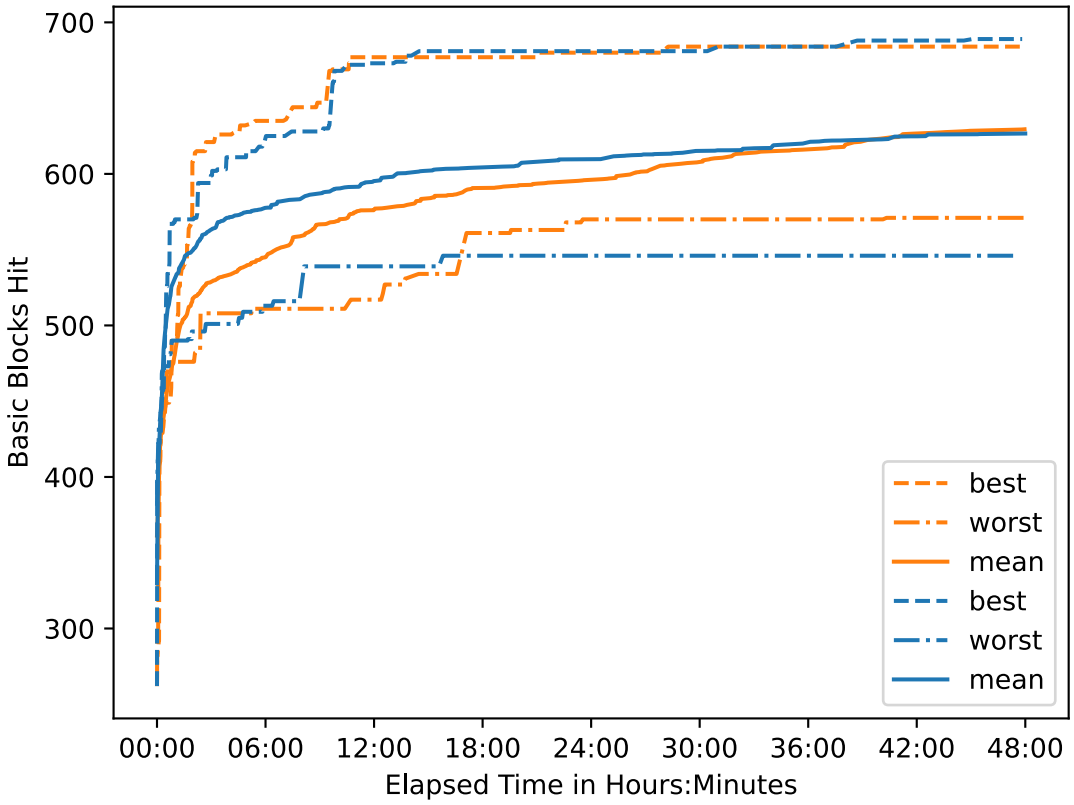


(a) Mean, Best, and Worst run

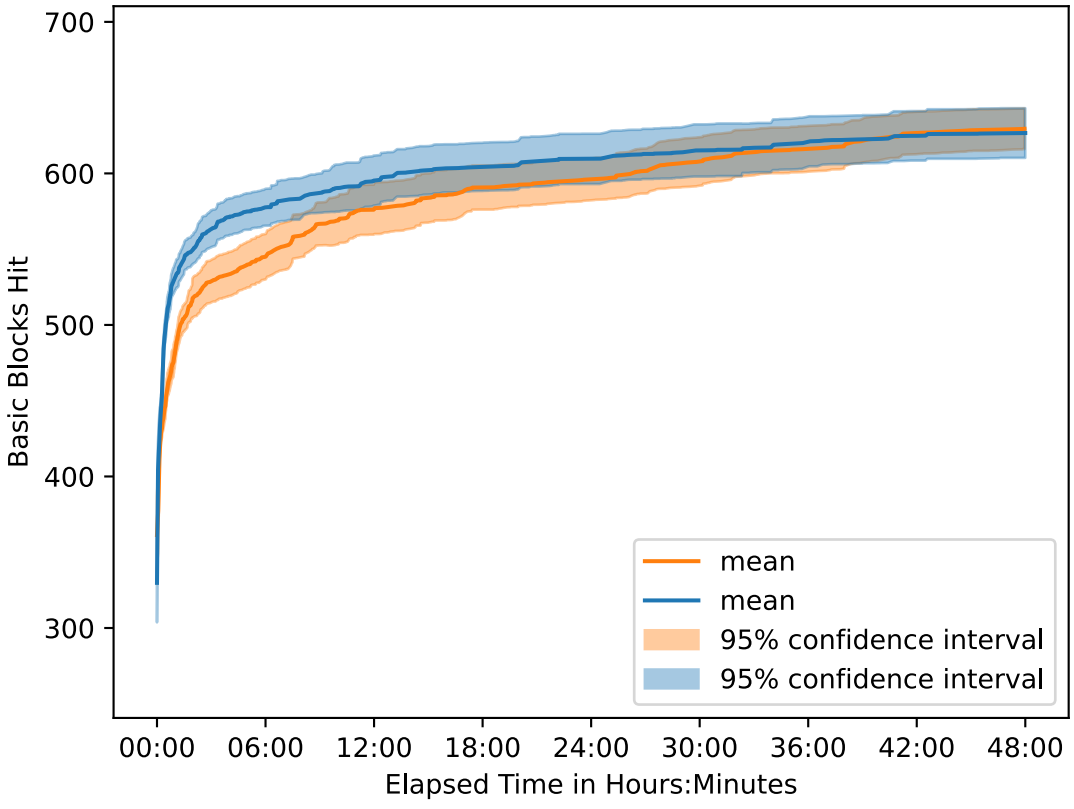


(b) Mean and confidence interval

Figure 6.1: Coverage comparison in basic blocks over time of AFLNet (blue) with SNAPP (orange) on open62541.

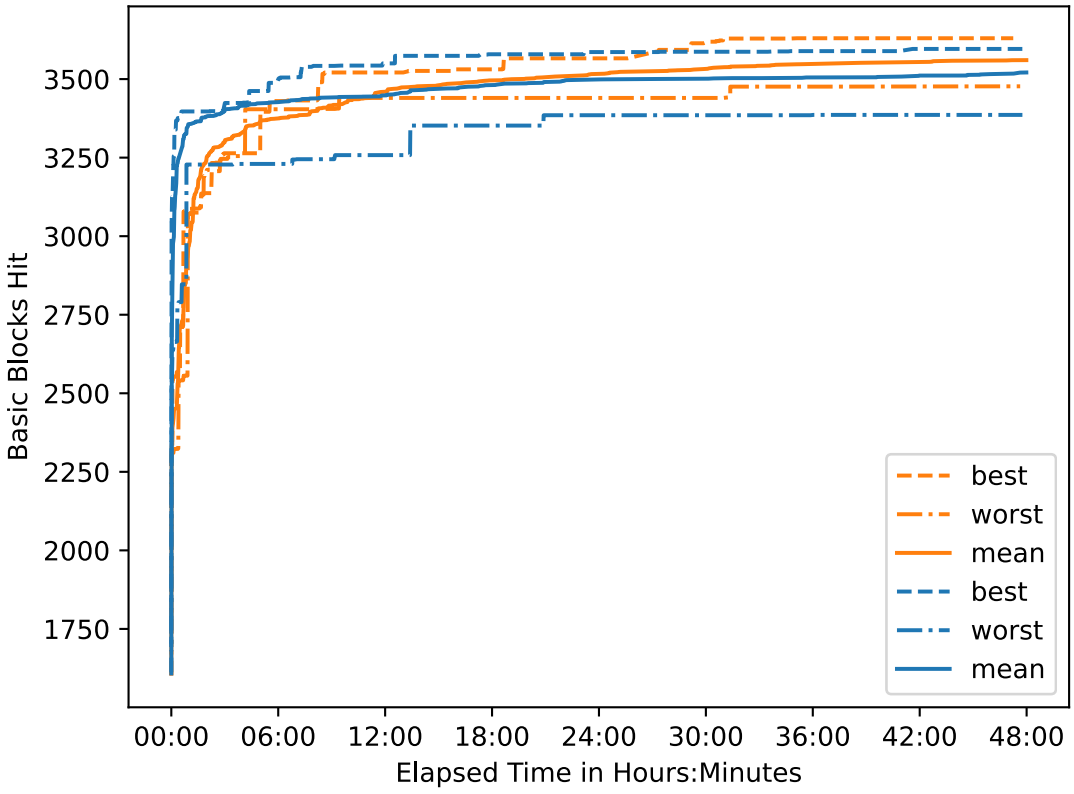


(a) Mean, Best, and Worst run

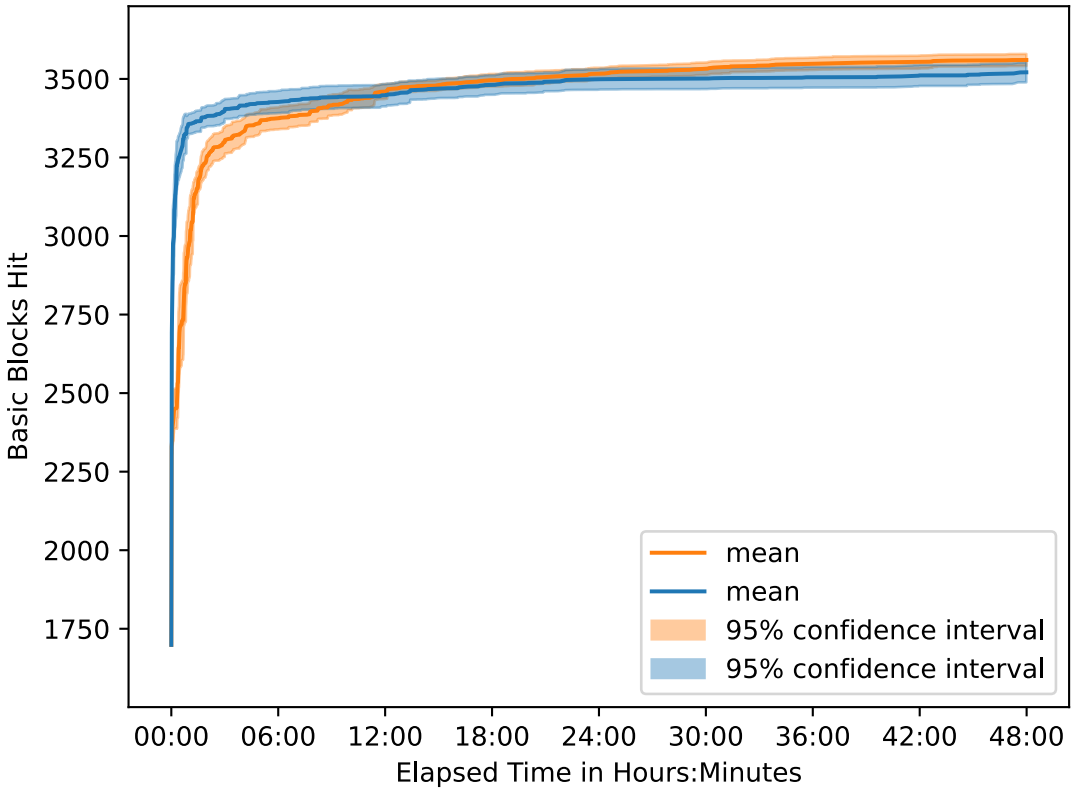


(b) Mean and confidence interval

Figure 6.2: Coverage comparison in basic blocks over time of AFLNet (blue) with SNAPP (orange) on lightftp.

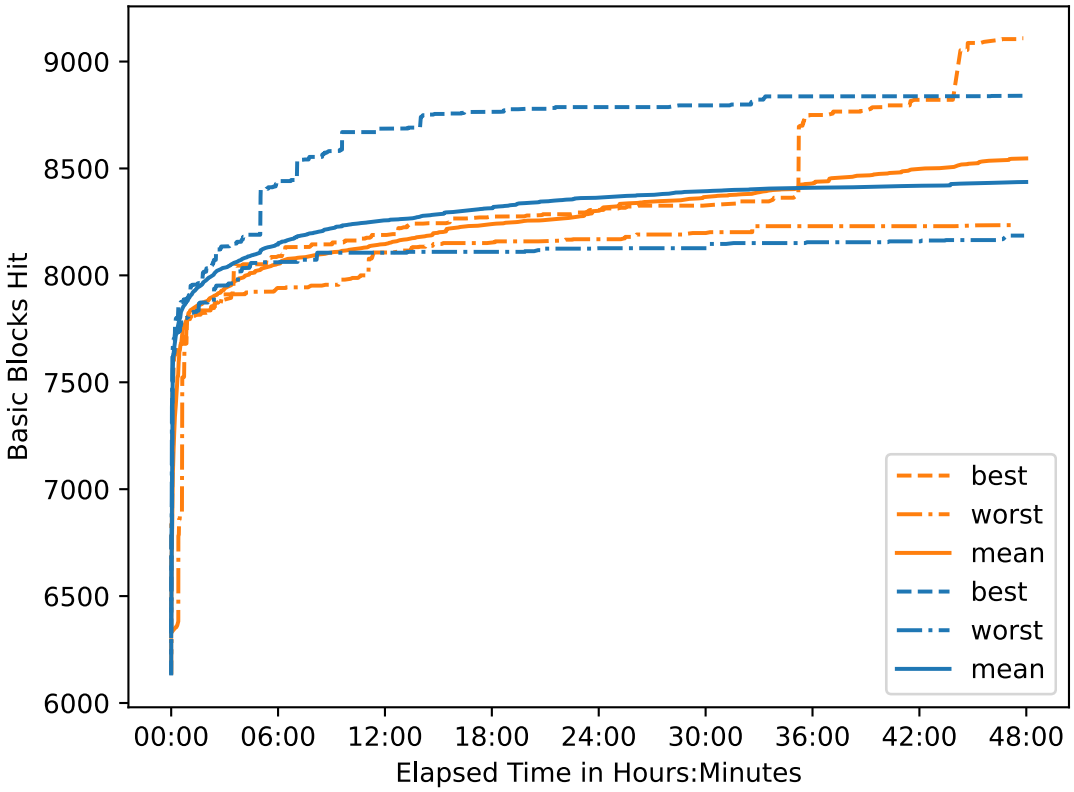


(a) Mean, Best, and Worst run

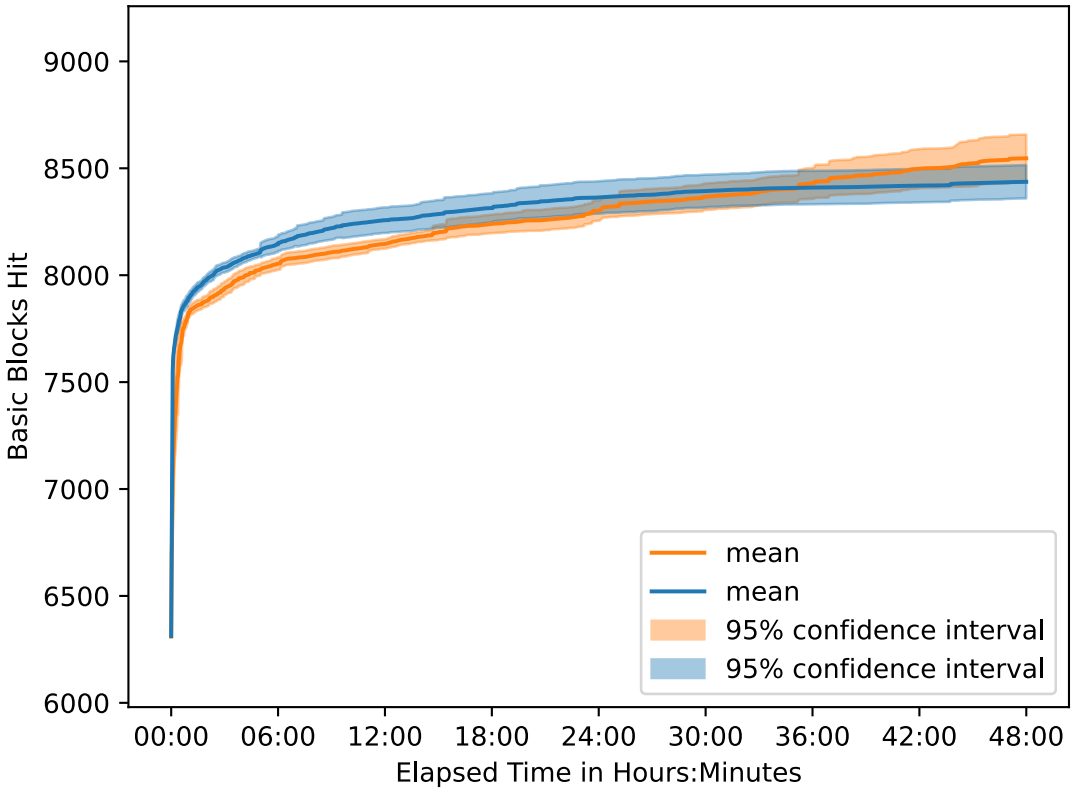


(b) Mean and confidence interval

Figure 6.3: Coverage comparison in basic blocks over time of AFLNet (blue) with SNAPP (orange) on live555.

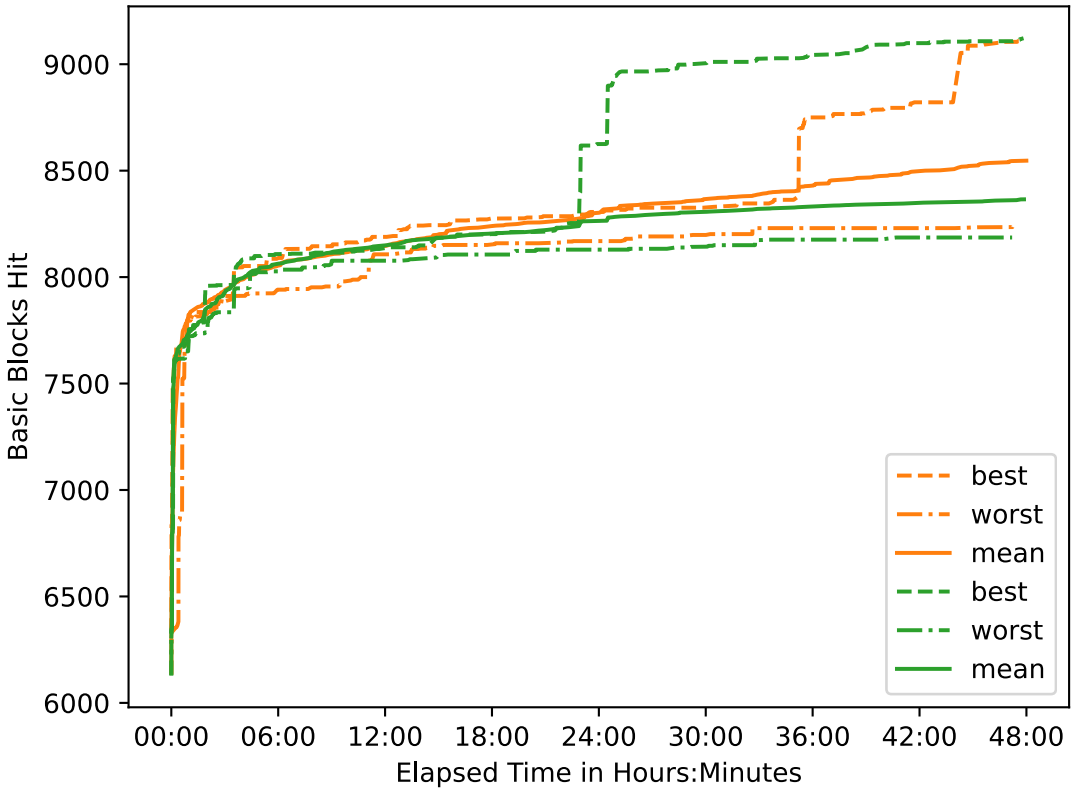


(a) Mean, Best, and Worst run

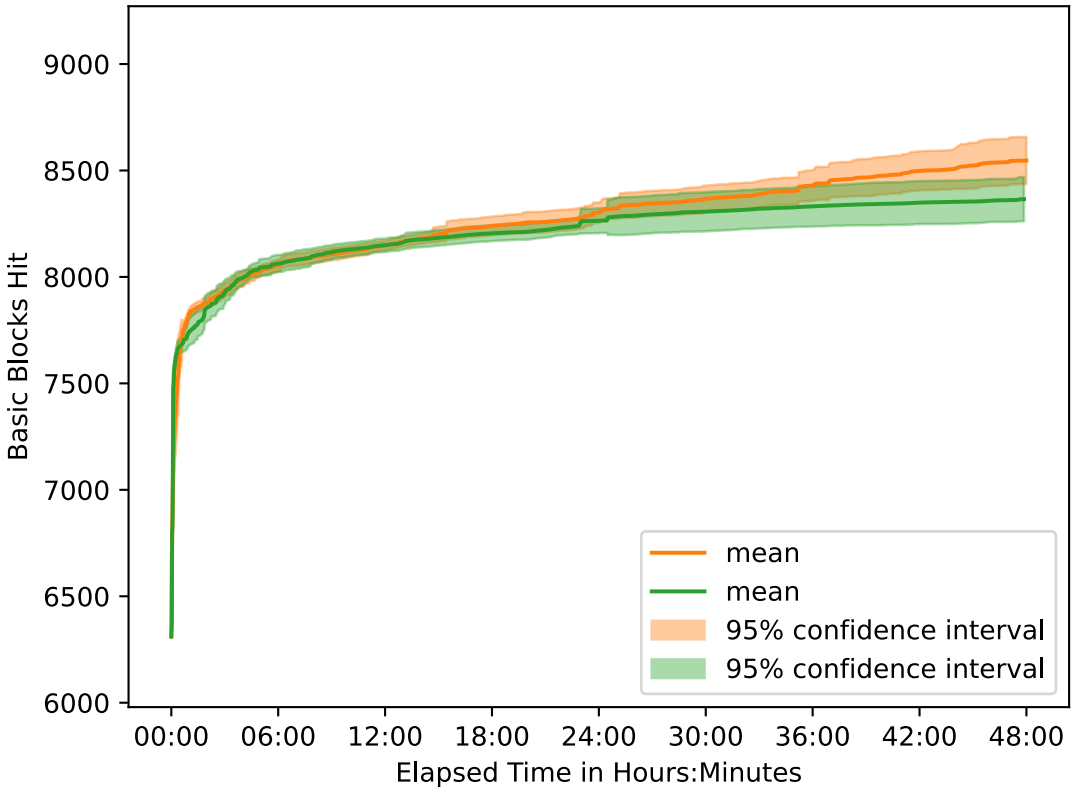


(b) Mean and confidence interval

Figure 6.4: Coverage comparison in basic blocks over time of AFLNet (blue) with FSNAPP (orange) on open62541 with the state forkserver enabled.

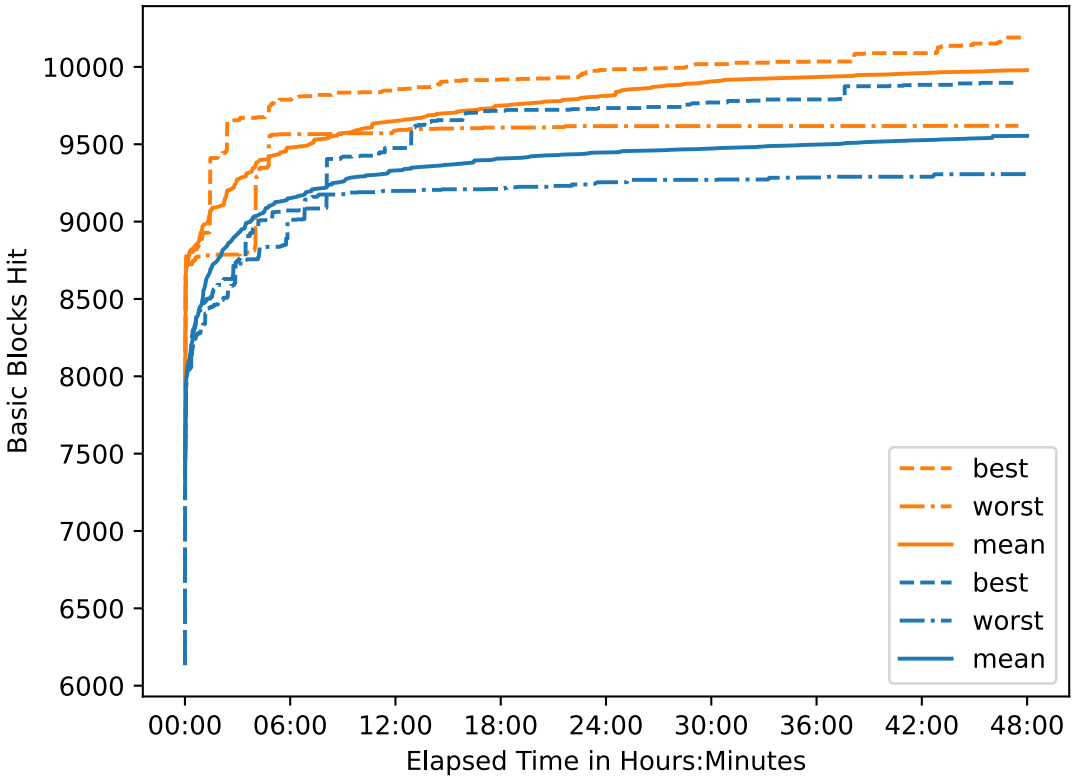


(a) Mean, Best, and Worst run

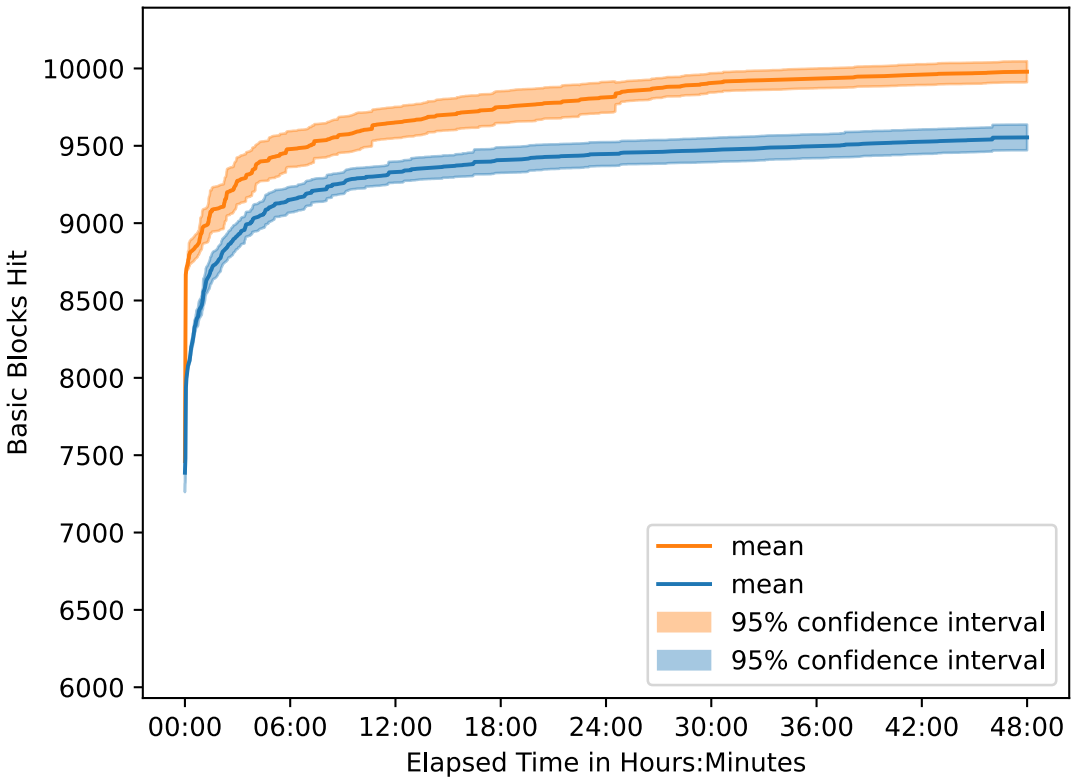


(b) Mean and confidence interval

Figure 6.5: Coverage comparison in basic blocks over time of SNAPP (green) with FSNAPP (orange) on open62541.

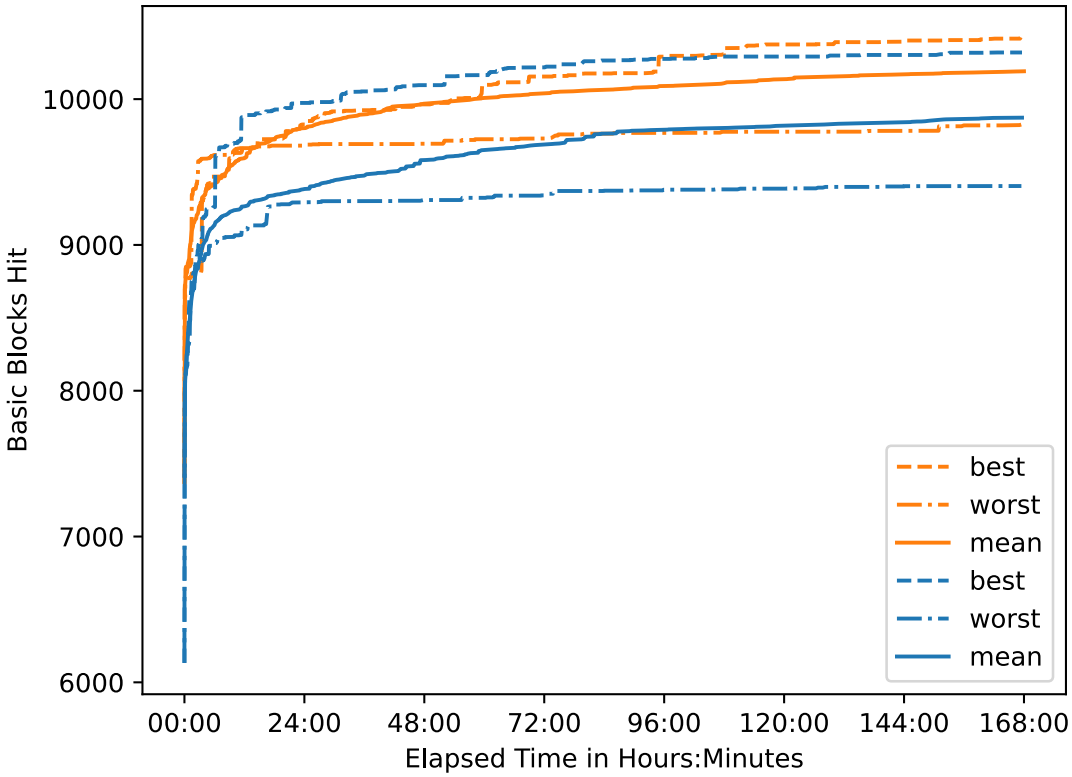


(a) Mean, Best, and Worst run

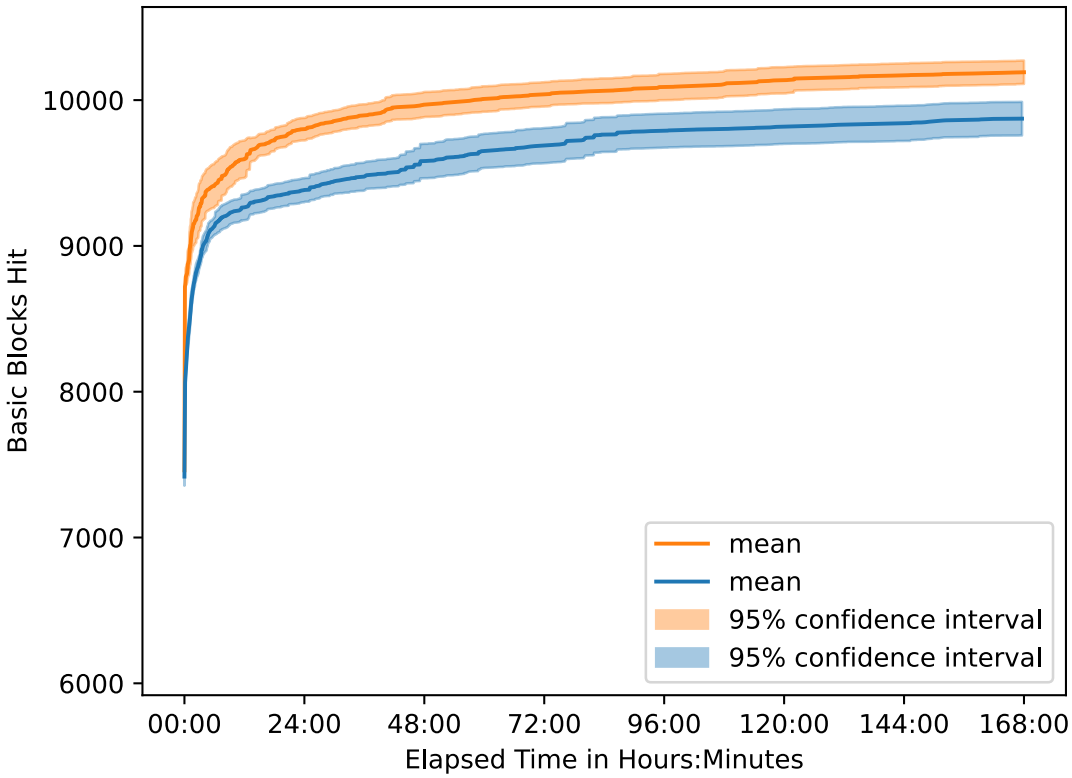


(b) Mean and confidence interval

Figure 6.6: Coverage comparison in basic blocks over time of AFLNet (blue) with SNAPP (orange) on open62541 with a bigger corpus.

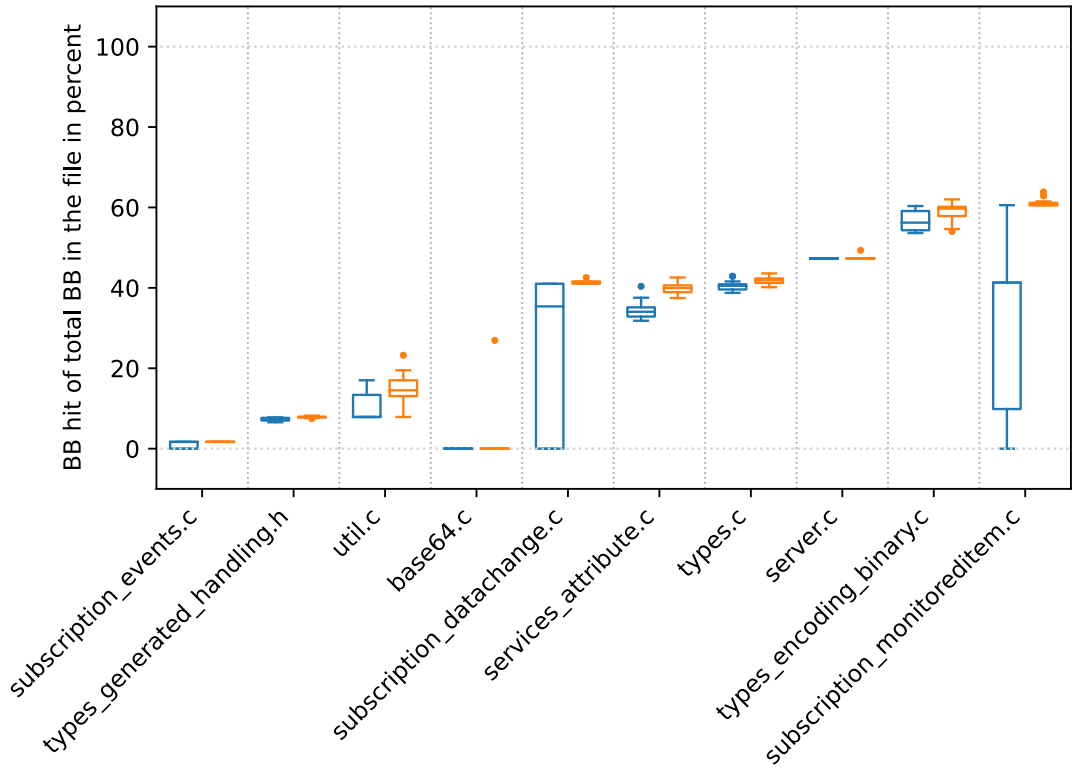


(a) Mean, Best, and Worst run

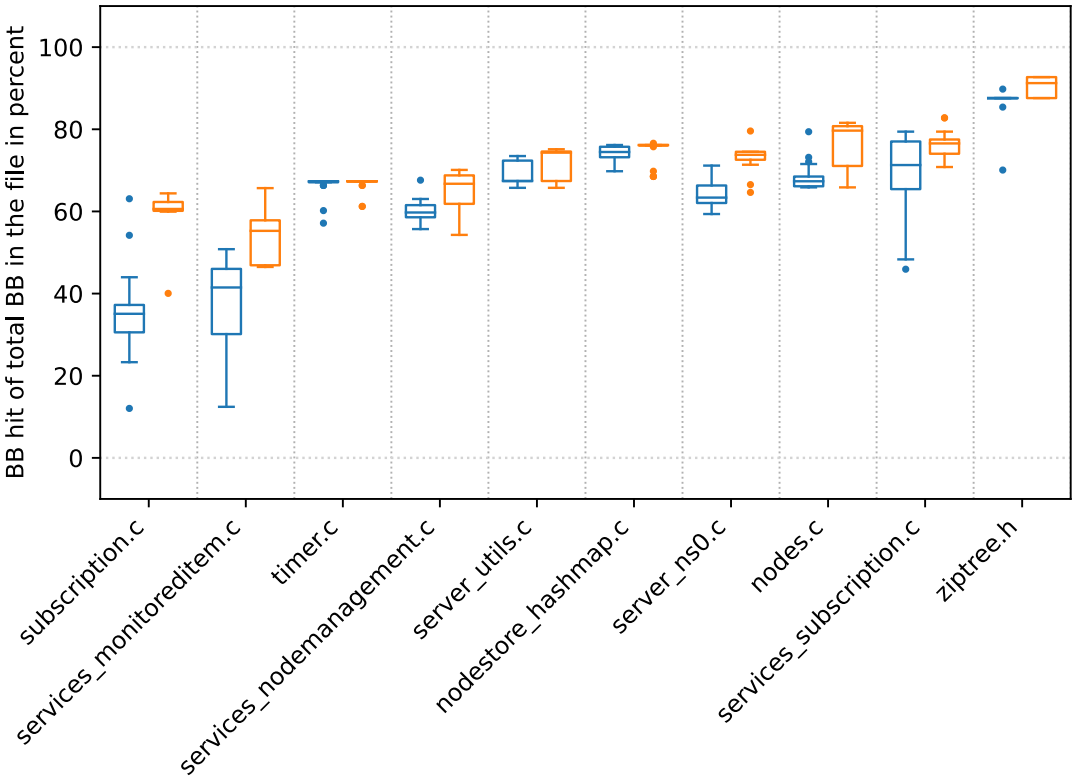


(b) Mean and confidence interval

Figure 6.7: Coverage comparison in basic blocks over time of AFLNet (blue) with SNAPP (orange) on open62541 with a bigger corpus over a duration of 7 days.

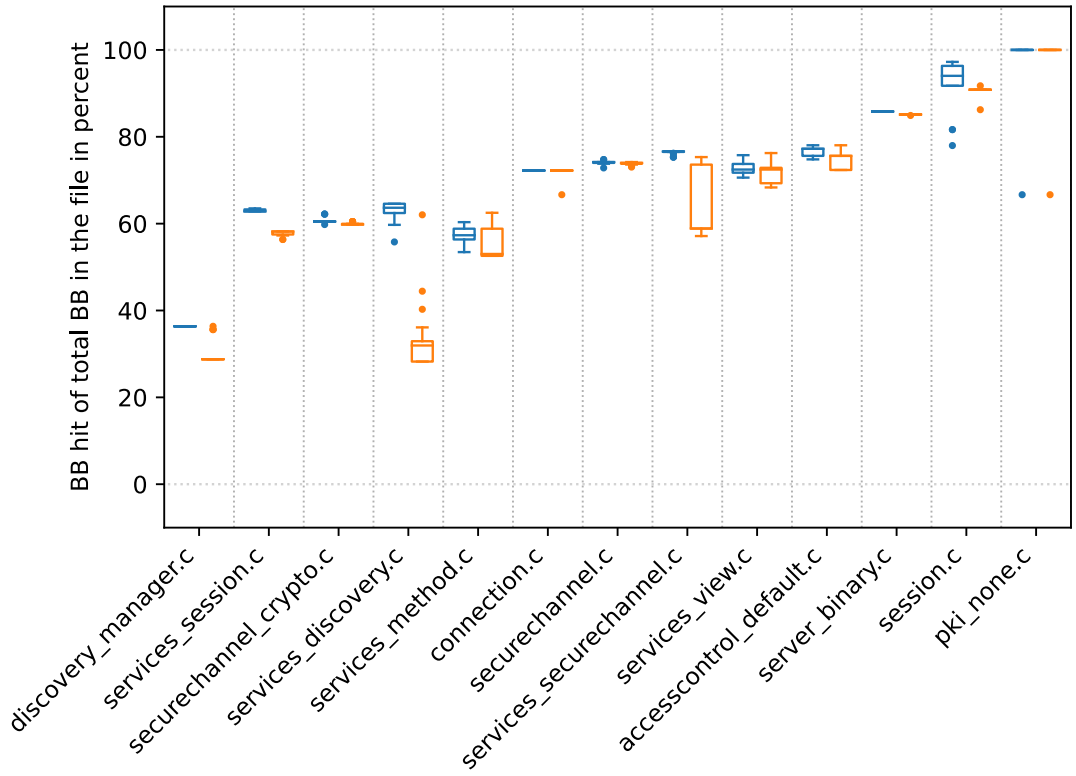


(a) Files where average coverage of AFLNet is worse than SNAPP



(b) More files where average coverage of AFLNet is worse than SNAPP

Figure 6.8: Final coverage in percent of BB across all runs by file for AFLNet (blue) and SNAPP (orange) on open62541 for the 7 day run. Files with the same basic blocks hit are omitted. Percentages relative to the total basic blocks per file.



(c) Files where average coverage of AFLNet is better than SNAPP

Figure 6.8: Final coverage in percent of BB across all runs by file for AFLNet (blue) and SNAPP (orange) on open62541 for the 7 day run. Files with the same basic blocks hit are omitted. Percentages relative to the total basic blocks per file. (cont.)

6.3 Crashes and Hangs

In order to conclude the evaluation we briefly discuss the results in Table 6.6. We also take a look at two bugs that were discovered and fixed before the actual evaluation runs.

6.3.1 False Positives

For open62541 only SNAPP found crashing inputs. However, these inputs are only able to crash the fuzzer-instrumented executable. When executed on the executable used to gather the coverage information, they do not result in a crash. This points toward a bug in the instrumentation logic of the fuzzer. Due to the difficulty of debugging the instrumented executable, the bug causing the crash could not be found. Since the bug only occurs on very few inputs, the results of the evaluation are still relevant.

The hangs found by AFLNet on open62541 were tested by randomly picking 5 inputs for each run. Testing every hang manually would have required too much time, since there are more than 4000 such inputs. For all samples, none of them actually caused the server to hang more than a second. Some inputs consist of very large inputs that naturally take longer to process.

For lightftp the hangs were sampled as well, by randomly picking 5 inputs for each run. This also did not yield any actual hangs.

The live555 evaluation yielded on average 263 crashes for AFLNet and on average 117 hangs for SNAPP. However, when testing a random sample of 5 inputs categorized as crashes by AFLNet for each run, no crash could be reproduced on either the fuzzer instrumented executable, or the coverage instrumented executable. The inputs categorized as hangs by SNAPP were evaluated like this as well. Most of the hangs could not be reproduced. Some of them however could be reproduced, but the cause of the hang was that the session id was fixed to the value 8888 for fuzzing and this caused an endless loop. The endless loop occurs, because live555 wants to generate a new session id, because the current one is already used. Since the value is fixed, however, this will always be the case, resulting in an endless loop. However, this is not an actual issue, since the value is only fixed for the fuzzing process.

6.3.2 live555 Crash

One of the hangs sampled from the SNAPP evaluation resulted in a crash on the actual executable. The hang in question is `id:000067,src:000340,time:37986884,op:havoc,rep:16`, in the results of the first evaluation run. The live555 executable crashes with a segmentation fault. When compiled with sanitizers, they identify the fault as a use-after-free, and the bug seems to correspond to the CVE 2019-7314.[1] When updating to the most recent version, the crash does not occur anymore.

6.3.3 Found Bugs

During the development of the fuzzer and the integration of the SUTs the following two bugs were found in open62541.

The first bug[2] does not crash the server. However, it violates the protocol specification, since it is possible to initiate a connection by starting with the OPN message, instead of a HEL message as required by the specification. The client simply does not configure its protocol parameters. The server then just assumes the default values and continues as if a HEL message

1 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7314
2 https://github.com/open62541/open62541/commit/a2c677cab669f2913dd50936ab1640da8a9760f3

was received. However, the specification requires that the communication is initiated with a HEL message [Fou17c].

The second bug[3] results in a crash of the server. By sending multiple messages in the same TCP packet where any message except the last results in an error, the server crashes, because it transitions into an error state, but still continues to process any remaining messages. This will then access already freed memory, resulting in the crash.

The first bug could not have been discovered by the fuzzers, because it does not result in a crash of the server or any kind of undefined behavior. The second bug was so severe, that it would have caused most of the inputs of the fuzzer to result in a crash. This would have hindered a sensible evaluation. Before running the actual fuzzing campaigns, both bugs were thus fixed with a pull request[4] to the open62541 repository.

3 https://github.com/open62541/open62541/commit/b74eadae948a730d63948b18fb854ee641ada787
4 https://github.com/open62541/open62541/pull/3831

7 Discussion

In this chapter we interpret the results of the evaluation presented in the previous chapter. We first answer each research question posed in Chapter 3 in Section 7.1. Afterwards, we discuss some additional insights gained from the evaluation data in Section 7.2. Further, we discuss some limitations of the current approach that were identified during the evaluation. In Section 7.4 we then discuss approaches related to this thesis, followed by proposals for future work in Section 7.5. Finally, we conclude the discussion in Section 7.6 by summarizing the overall findings.

7.1 Answering the Research Questions

We now use the evaluation data presented in Chapter 6 in order to answer the research questions posed in Section 3.3.

7.1.1 RQ1: Can the state identification process be automated by using already provided coverage information?

In Chapter 4 we introduced new mechanisms in order to extract state information from already provided coverage feedback. In order to answer RQ1, we presented the evaluation runs in Chapter 6 comparing the coverage achieved over time on multiple fuzzer-SUT-corpus combinations.

We repeat Table 6.7 here for convenience. The results presented in Table 7.1 suggest that in most cases SNAPP performs at least as well as AFLNet. The only case where SNAPP performs worse is for open62541 on a small corpus. Although the result is statistically significant ($p < 0.05$), the overall decrease in coverage is only -4.05%. It should be noted however, that in the 48 hours the test ran for, AFLNet fuzzed the SUT nearly twice as much as SNAPP (see Table 6.6). Since SNAPP's implementation uses a very rudimentary synchronisation measure, the speed can most likely be improved such that SNAPP runs as fast as AFLNet. This in turn could reduce the difference in achieved coverage to a statistically not significant amount such that the fuzzers achieve comparable performance. The results for FSNAPP, which implements

		Basic Block Coverage		
		% Incr.	p-value	Figure
SNAPP vs AFLNet	open62541	−4.05	0.038	6.1
	lightftp	0.31	0.946	6.2
	live555	1.11	0.091	6.3
FSNAPP vs AFLNet	open62541	1.04	0.222	6.4
FSNAPP vs SNAPP	open62541	5.31	0.005	6.5
SNAPP large vs AFLNet	open62541	4.44	< 0.001	6.6
SNAPP long vs AFLNet	open62541	3.22	< 0.001	6.7

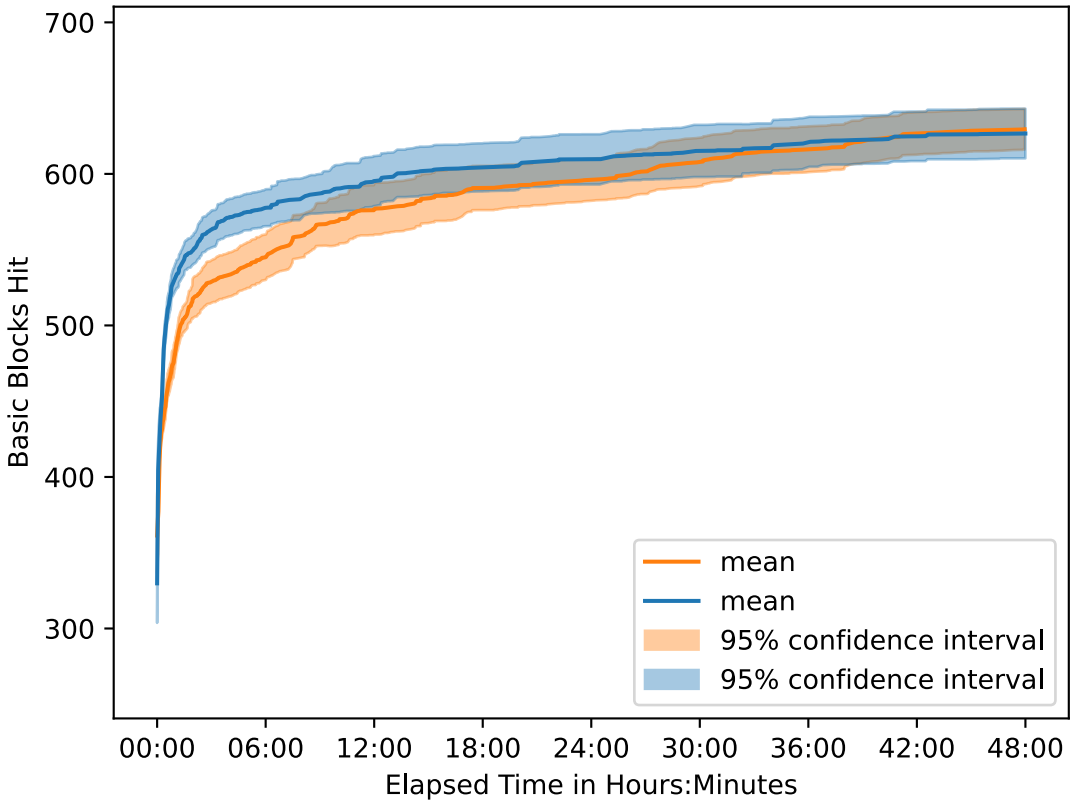
Table 7.1: Mean coverage increase and statistical significance (p-value) when comparing SNAPP and FSNAPP to AFLNet.

a simple snapshotting mechanism (see section 4.6), show that using snapshot mechanisms is a feasible approach in order to improve the performance of SNAPP. FSNAPP achieves better coverage than AFLNet as seen in Figure 6.4 and Table 7.1. Overall, we can conclude that the automatic state extraction used by SNAPP can successfully replace the manual approach used by AFLNet, since it performs at least as well as AFLNet on all but one SUT-corpus configuration.

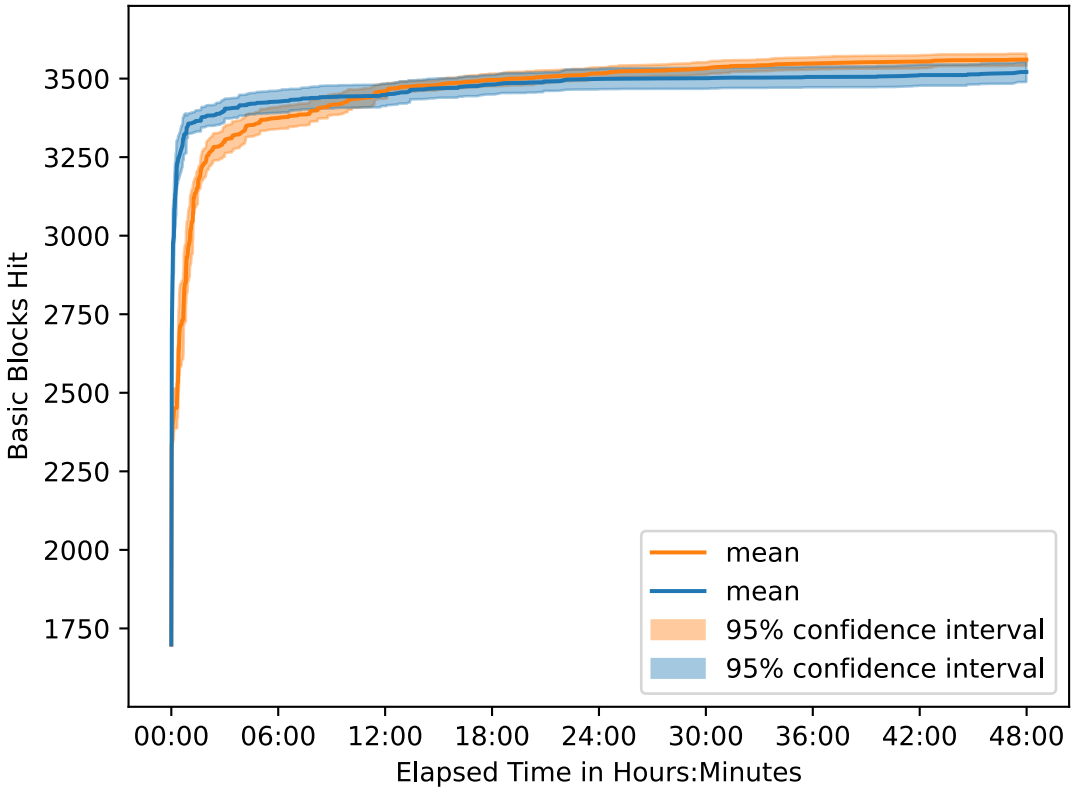
7.1.2 RQ2: How effective is a fuzzer using an automated state identification process compared to the manual approach of AFLNet if measuring performance by using coverage achieved as metric?

We now compare the two approaches SNAPP and AFLNet in more detail. Recall, that in order to directly compare SNAPP to AFLNet we ran a fuzzing campaign on each SUT also used by Pham et al. in the AFLNet paper [PBR20b]. For lightftp this resulted in almost identical coverage achieved on average after 48 hours as seen in Table 6.6, Table 7.1, and Figure 7.1a. Since we used the same corpus and the same SUT version for both lightftp and live555 as Pham et al. [PBR20b], we can also indirectly compare the achieved results to AFLNwe and boofuzz.

AFLNet achieved a 121% coverage increase compared to AFLNwe (see Table 7.2), so we should expect similar results when comparing SNAPP to AFLNwe. The same reasoning can be applied to the results achieved for boofuzz. A direct comparison should be made in the future, but due to time constraints we limit ourselves to the indirect comparison for now. It should also be mentioned that lightftp is a comparatively small SUT, having only a total of 2 236 lines of code. This means that the complexity of the SUT is not as high as for example open62541 with its 54 904 lines of code, and as such it is easier for a fuzzer to achieve high coverage.



(a) lightftp



(b) live555

Figure 7.1: Coverage comparison in basic blocks over time of AFLNet (blue) with SNAPP (orange)

		Branch Coverage			Statement Coverage		
		% Incr.	$\hat{A}_{12}$	p-value	% Incr.	$\hat{A}_{12}$	p-value
AFLNet vs AFLNwe	lightftp	121.06	1.000	< 0.001	79.45	1.000	< 0.001
	live555	3.40	0.335	0.076	2.44	0.228	0.003
AFLNet vs boofuzz	lightftp	57.73	1.000	0.026	49.72	1.000	0.026
	live555	64.13	1.000	0.026	62.09	1.000	0.026

Table 7.2: Mean coverage increase (%Increase), effect size ($\hat{A}_{12}$), and statistical significance (p-value) when comparing AFLNet to boofuzz and AFLNwe, respectively. A Vargha-Delaney $\hat{A}_{12}$ measure above 0.71 indicates a large effect size in favor of AFLNet. Statistical significance is computed using the Mann-Whitney U test. Adapted from [PBR20b]

For live555 the achieved coverage for SNAPP and AFLNet is also very similar as seen in Table 6.6, Table 7.1, and Figure 7.1b. This can again be indirectly compared against the results achieved by boofuzz and AFLNwe on live555. However, for live555 the results should be interpreted with a grain of salt, since the stability (see Section 6.2.1) of the fuzzer was low, i.e. 3% for AFLNet and 30% for SNAPP. This means that the executable behaved in a very nondeterministic way, potentially hindering the fuzzer from progressing in a meaningful way. This assumption is reinforced by the coverage curves in Figure 7.1b. The coverage only marginally increases after the first 6 hours, suggesting that the fuzzer had difficulty finding new interesting inputs. Also, compared to all other SUTs, the best and worst runs of live555 are not far apart, which is also reflected in the confidence interval being small compared to the other SUTs. In Section 7.3 we discuss possible causes for this nondeterministic behavior.

Finally, the runs performed on open62541 show, that when choosing an extensive corpus that contains a template for all message types the SUT can process, SNAPP is able to outperform AFLNet as seen in Figure 6.6 and Figure 6.7. We discuss the meaning of the corpus choice in more detail in the following section, answering RQ3.

Overall, we can conclude that SNAPP is able to achieve similar or better coverage on stateful SUTs compared to AFLNet. Most importantly, SNAPP does so without needing a manual specification of the response code extraction function. Additionally, if the SUT does not supply much information via response codes, the approach used by SNAPP can be used to still identify states by using already supplied coverage feedback. This approach also works, even if the source code of the SUT is not available, and it is only available as a binary. Although in that case the derandomisation and synchronisation injection becomes more difficult, since the binary has to be modified.

7.1.3 RQ3: Does the seed corpus selection make a difference in achieved coverage when fuzzing stateful software?

Discovering new states often corresponds to finding a new sequence of messages in which each message passes the parsing stage in order to be processed further by the SUT. These sequences are found by combining already known messages into a new sequence, or by mutating existing sequences with the mutation operations described in Section 4.5. Since this requires already known messages, the first step before mutating a sequence is finding new messages that pass the parsing stage. This can be done either by mutating existing messages, or by supplying the fuzzer more initial testcases that contain valid messages. An obvious hypothesis is therefore, that adding more message types to the initial corpus should increase the performance of the fuzzer.

	Basic Block Coverage		
	% Incr.	p-value	Figure
AFLNET	12.94	< 0.001	7.2a
SNAPP	22.95	< 0.001	7.2b

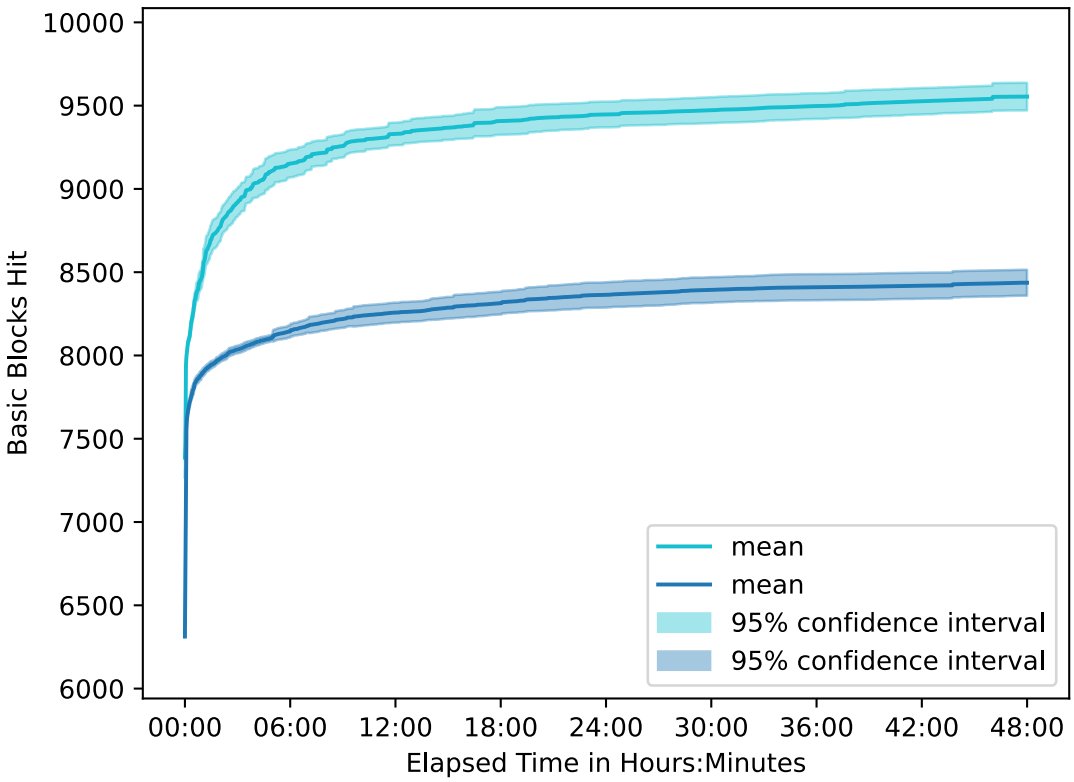
Table 7.3: Mean coverage increase and statistical significance (p-value) when comparing small and large corpus runs of AFLNet and SNAPP respectively.

In order to verify this hypothesis, we ran the open62541 SUT on two corpora with different sizes (see Table 6.3 and Table 6.4). For both AFLNet and SNAPP the larger corpus results in a significant increase of the achieved coverage as seen in Table 6.6. These results are also plotted once more in Figure 7.2, directly comparing AFLNet and SNAPP respectively on a small and large corpus. Interestingly, the increase in percent for SNAPP is almost 1.8 times as high than that of AFLNet, suggesting that SNAPP can handle a more diverse initial corpus better. We discuss possible reasons for this in Section 7.2.

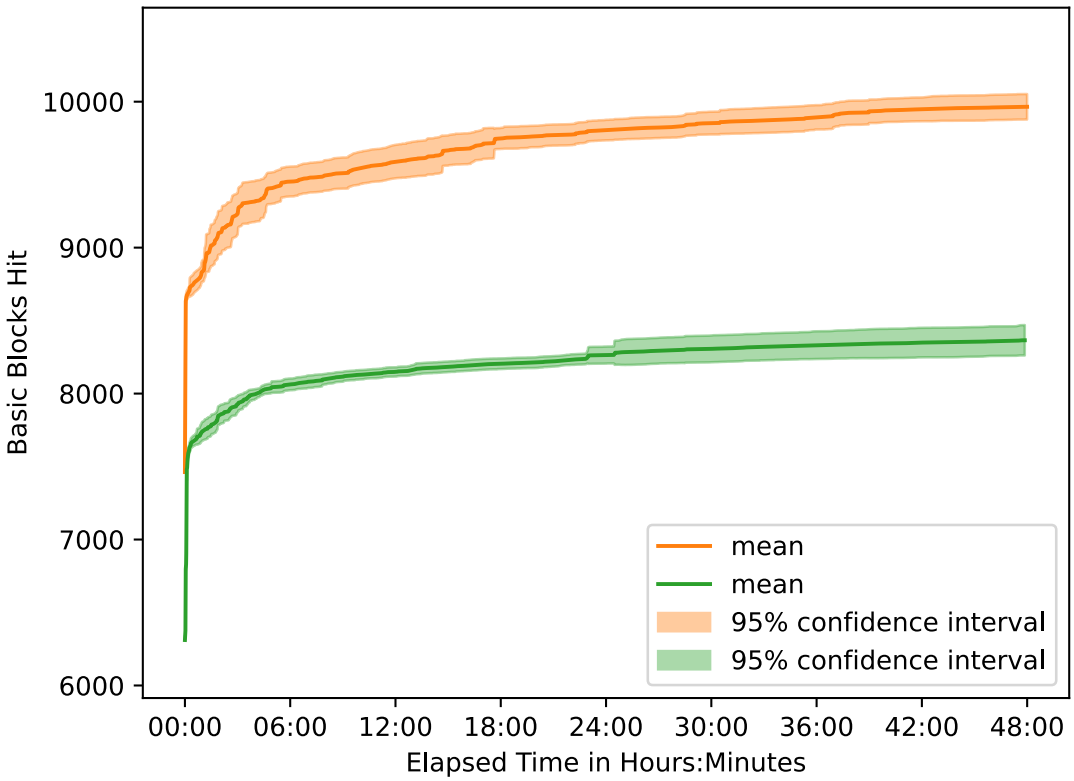
As we have seen, the achieved coverage can be significantly improved by supplying a better initial corpus to the fuzzer that contains a more diverse set of message types. This suggests that the fuzzer performance could possibly be increased by using a generational approach in order to generate valid messages for the fuzzer. We discuss this further in Section 7.5.

7.1.4 RQ4: Could program state snapshotting improve fuzzing speed and as such effectiveness?

Chen et al. [ClV19] propose an approach using a stateful forkserver in order to fuzz stateful SUTs. Their work is covered in Section 7.4. We adapted a similar forkserver approach for state



(a) AFLNet on open62541 with a small corpus (blue) with a large corpus (cyan)



(b) SNAPP on open62541 with a small corpus (green) with a large corpus (orange)

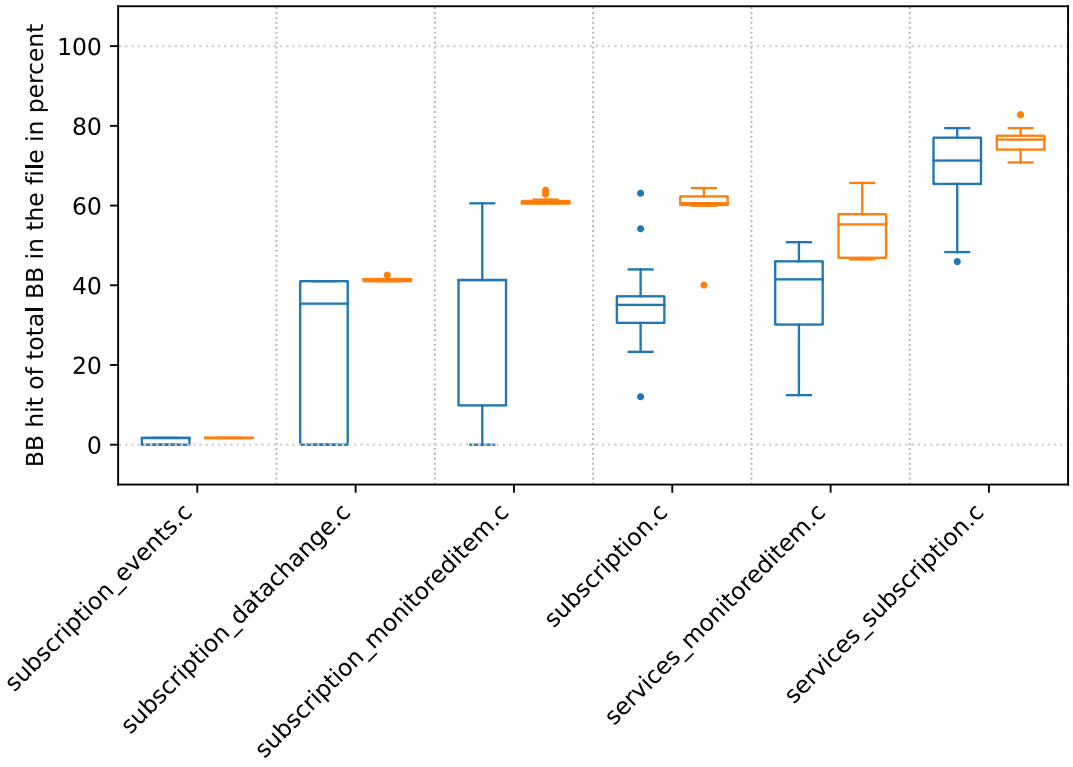
Figure 7.2: Coverage comparison in basic blocks over time.

snapshotting (see Section 4.6). The results seen in Table 6.6, Table 7.1, Figure 6.4, and Figure 6.5 show that the snapshotting mechanism increases the performance of SNAPP. FSNAPP (the SNAPP version with enabled snapshotting) is able to achieve 5.31% more coverage compared to SNAPP. This increase in coverage is statistically significant and shows that a snapshotting approach can be used to increase performance. However, since only a rudimentary version without any optimizations was implemented due to time constraints, the performance increase can possibly be further increased. Furthermore, problems with correct restoring of snapshots arose during implementation that are further discussed in Section 7.3. Overall we can conclude that snapshotting techniques are able to increase fuzzer performance and are worth further investigation (see Section 7.5).

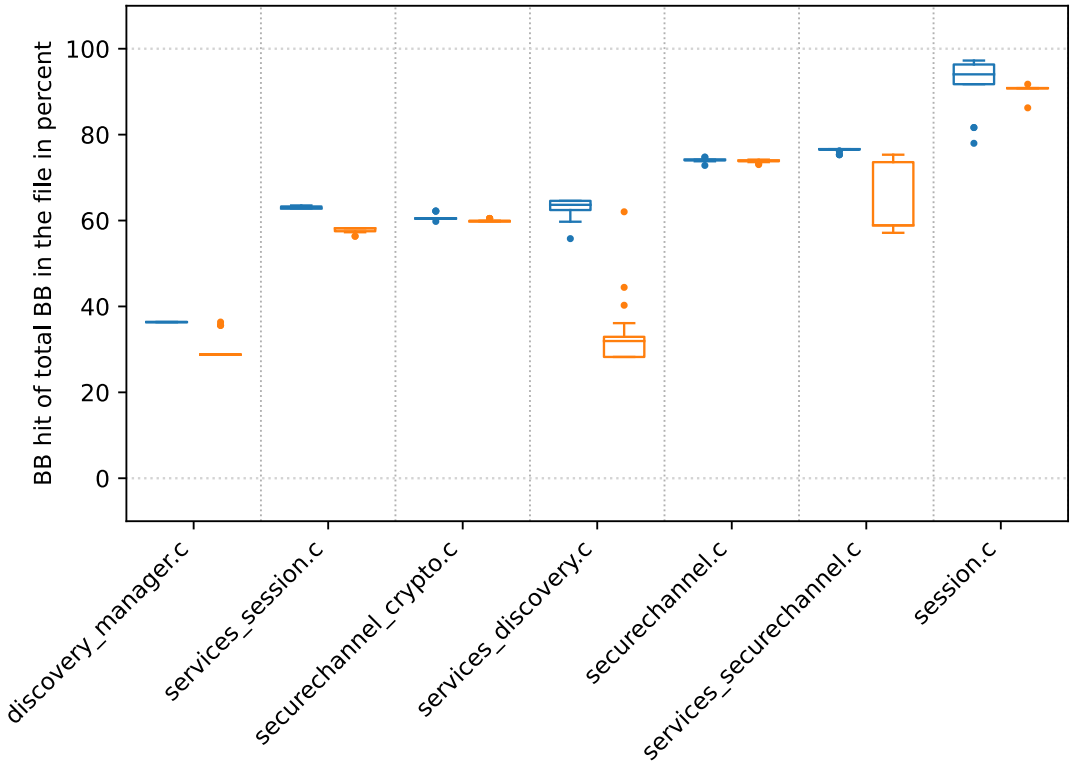
7.2 Additional Insights

By comparing the final achieved average coverage per source file on open62541, some additional insights can be gained. The three plots in Figure 6.8 depict a box plot. Each source file has its own column. The left (blue) and right (orange) box plot depicts the final coverage achieved for all twenty runs with the big corpus over 7 days on open62541 of AFLNet and SNAPP respectively. Files with no difference in basic blocks over all runs are omitted.

An interesting observation is that all files handling a part of the subscription service exhibit a high variance for AFLNet compared to SNAPP. The files in question are plotted together once more in Figure 7.3. The subscription service model is highly stateful. The standard lists five different states with a total of 27 different transitions. In order to model this complex statemachine, AFLNet would need a very detailed message encoder/decoder akin to a small client. Implementing such a model is error prone and requires more work in order to adapt new protocols. The implemented parser thus only decodes messages and extracts state IDs up to the SecureChannel layer. AFLNet can thus not identify any states that are observed in a higher layer. SNAPP however, is able to fuzz these deep states more effectively as seen in Figure 7.3 because of its automatic state identification component. AFLNet seems to however perform slightly better on files that are reachable without deep state dependencies. As seen in Figure 7.3b, for the files handling the session service, the securechannel service, and the discovery service the coverage results are in favor of AFLNet. All of these services are directly reachable from the states modeled for AFLNet, and the inter-message dependencies are not as complex as those for the subscription services. Overall, this indicates that SNAPP is able to successfully fuzz deeper states of the open62541 target than AFLNet.



(a) Only files concerning the session service are shown.



(b) Only files concerning the securechannel, session and discovery service a re shown.

Figure 7.3: Final coverage in percent of BB across all runs by file for AFLNet (blue) and SNAPP (orange) on open62541 for the 7 day run. Percentages relative to the total basic blocks per file.

7.3 Limitations

We now discuss limitations encountered with the current approach of AFLNet and SNAPP. First, we look at fuzzer stability. Afterwards we consider speed concerns, and finally we examine problems of the current snapshotting approach.

7.3.1 Stability

As already mentioned in Section 7.1, the stability (see section 6.2.1) of the live555 SUT is very low. In order to understand why this is the case, we have to consider what the live555 SUT does. The live555 code itself is mostly deterministic. It is single threaded, and uses very little random number queries. The random seed and clock could have been made deterministic as well as done for open62541, but we chose not to do so, in order to achieve comparable results to Pham et al. for AFLNet [PBR20b].

The actual problem arises when considering the syscalls live555 makes. Since live555 implements the streaming protocol RTSP, the code makes extensive use of file operations. Since the execution times and behavior of these syscalls depend on the hardware configuration and kernel behavior, a single input can behave differently if it is sent more than once. The same problem exists for syscalls like `send` and `recv` that handle the network messages. In order to alleviate this problem, the syscalls would most likely have to be intercepted by the fuzzer in order to introduce deterministic behavior. We discuss possible approaches further in Section 7.5.

7.3.2 Speed

The current implementations of AFLNet and SNAPP both perform an order of magnitude slower than conventional fuzzers like AFL or libFuzzer. Although this speed loss is compensated by more efficient mutations, there is still potential for increased speed as shown by the forkserver experiment. In the fuzzing community the consensus is, that fuzzers should be as fast as possible, due to the nature of random testing [LLV20]. The reasoning being, that more inputs tested result in a higher probability of finding bugs.

7.3.3 State Forkserver

The state forkserver implemented for this thesis did not work on the live555 and lightftp SUTs. The reason for this is that both SUTs use file operations. The implemented forkserver simply forks the SUT after it has received the messages in a sequence (see Chapter 5). It is important to note that the open file descriptors persist across such forks. For the network connection

this is intended, and works as expected. However, file descriptors for actual files on disk have additional state, like the position of the cursor inside the file. Since this additional state data is not reset in between fork calls, the program behaves in an unexpected way for every fork but the first.

For example, consider that the current snapshot has one open file with the cursor set at the beginning. If the first forked instance then processes an input that sets the file descriptor of an open file to the middle of the file, this change will persist for the next forked instance. The second forked instance will then start from a point where the cursor in the open file is in the middle, which is not the correct state as it would be expected. This problem also manifests itself, if any files are created or deleted after the snapshot point. We discuss possible solutions for this problem in Section 7.5.

7.4 Related Work

In this section we discuss work related to the problem of fuzzing stateful software. We first cover existing blackbox approaches in Section 7.4.1 and greybox approaches in Section 7.4.2. Afterwards, we briefly discuss related approaches for state snapshots in Section 7.4.3. In Section 7.4.4 we then explain the relevance of symbolic execution for stateful fuzzing. Finally, we cover approaches to target oriented fuzzing in Section 7.4.5 that are relevant for guiding a fuzzer to locations that were identified as interesting.

7.4.1 Blackbox State Identification

As already mentioned in Section 3.1, the state identification process using blackbox approaches has seen extensive research recently [Dou+12; RP15; Fit+20; AGP19; Ma+16; Gas+15]. The approach by Doupé et al. [Dou+12] uses a similar insight to the one introduced in Chapter 4. Their work focuses on blackbox testing of web applications. As such, they send requests to the web application and observe the responses, parsing them in order to extract relevant information. They make the observation that when sending an identical request twice and getting a different response, the state of the web application must have changed. The difference to the state identification algorithm introduced in Chapter 4 is, that our approach uses coverage feedback of the SUT and as such is independent of the SUT, as long as it can be instrumented to provide coverage feedback. In contrast, their approach is specific to web applications and uses a parser that extracts information about the links contained in a web page. This information is then used similarly to the message behavior defined in Definition 4.2.

Other approaches can be categorized into active and passive state machine learning. Active state machine learning approaches like those of De Ruiter et al. [RP15] and Fiterau-Brostean et al. [Fit+20] use interaction with the SUT to gain information used to infer a state machine. Passive state machine learning approaches like the Pulsar fuzzer by Gascon et al. [Gas+15] only have access to a fixed set of data that is supplied to the algorithm like a set of network captures of the relevant protocol in order to extract the state machine.

Finally, there are also model based approaches like boofuzz [Per20], Peach [Tec20], or the work by Ma et al. [Ma+16]. These approaches usually require the user to specify a message and/or state model for the fuzzer to start fuzzing. In contrast to these approaches, our approach avoids manual specification of a state machine model, and only requires a simple parser that is able to split an input string into individual requests. Also, in general, greybox fuzzers like our approach have an advantage over blackbox approaches, since the obtained coverage feedback allows for more fine-grained steps during mutation.

7.4.2 Stateful Greybox Fuzzing

There already exist greybox approaches that consider the stateful nature of SUTs. Steelix by Li et al. [Li+17] uses state information of the program extracted by static analysis and instrumenting the SUT. The state information Steelix extracts concerns the comparison progress on locations identified by the static analysis. Further, Wang et al. [Wan+20] use static analysis and feed the obtained results to the fuzzer, guiding it to operation sequences that are more likely to exercise use-after-free behavior. Both of these approaches regard the fine-grained state space of the program for single inputs in contrast to our approach, which tries to find states that are relevant for dependencies between multiple inputs.

Lastly, Aschermann et al. [Asc+20] recently developed a fuzzer called IJON. Their fuzzer uses source code annotations in order to give the fuzzer feedback in situations, where coverage feedback is of no use. For example, if the state is contained entirely in memory, but execution paths are the same until a relevant change to the memory region has been made, the fuzzer will not get any new feedback by only observing coverage information. Their solution in this case is to provide additional feedback to the fuzzer by inserting an annotation that for example returns the memory contents to the fuzzer, making changes to that region observable for the fuzzer. However, this requires manually inspecting the source code of the program in order to identify where to put annotations.

7.4.3 State Snapshots

Similar to the approach by Chen et al. [ClV19], we showed that using program state snapshotting can improve fuzzing performance, because restoring the snapshot is faster than restoring the state by sending the required inputs. The work by Dong et al. [Don+20] follows a similar approach. Their use case is specific to android app testing. Because of the testing environment, it is difficult to save input sequences to restore a state. Instead, they use the snapshotting mechanisms of virtualbox, saving an entire virtual machine state and restoring it later on if needed. Key difference in the mutation strategy is, that mutations do not start from the same state for each mutation done, but are performed on top of each other. The states are only used as means to travel back in time to a previously observed state that was considered interesting, if the fuzzer reaches a dead end.

7.4.4 Symbolic/Concolic Execution

Symbolic execution describes the process of using a constraint solver, in order to find a concrete input that exercises a specific part of the program. This approach can be categorized as whitebox fuzzing, since the program is not executed, and full source code knowledge is needed. Concolic execution on the other hand usually combines symbolic execution with actual executions of the program. A noteworthy example for concolic execution is the angr framework [Sho+16].

Angr is used by Driller, a fuzzer developed by Stephens et al. [Ste+16]. Traditional fuzzing often has difficulties trying to pass complex checks in the SUTs code, while concolic execution suffers from path explosion when used alone. By using concolic execution only for checks where the traditional fuzzers has difficulties, Driller is able to mitigate the weaknesses of each approach. Although their work does not directly consider the challenges in stateful fuzzing, it is nevertheless relevant. Their work can be used to more effectively find valid inputs, which are essential for efficient sequence mutations as we discuss in Section 7.5.

Further work in this direction includes QSYM by Yun et al. [Yun+18], which improves efficiency over driller by loosening the strict soundness requirements of symbolic execution, and REDQUEEN by Aschermann et al. [Asc+19]. REDQUEEN uses an approach leveraging what the authors call “input-to-state correspondence”, meaning that parts of the input often end up in the program state unmodified. However, their approach only tackles the problem of overcoming difficult checks for single inputs similar to Driller and QSYM, in contrast to our approach, which tries to find sequences of inputs.

7.4.5 Target Oriented Fuzzing

Target oriented fuzzing has been successfully employed to guide fuzzers to specific code locations. Target oriented fuzzing can for example be used to guide the fuzzer to newly introduced code in a repository. By only fuzzing the code difference, the available resources are concentrated on previously unfuzzed code, instead of using them to fuzz code that has already been fuzzed.

AFLGo by Böhme et al. [Böh+17] uses an annealing-based power schedule in order to assign more power to seeds that are closer to one or more desired target locations. AFLGo generates inputs without knowledge of the SUT's input structure. The TOFU fuzzer developed by Wang et al. [WLR20] identifies this as a problem. They reason that random changes to a valid input can change the input in such a way, that it is invalid and as such increases the distance from the target location. Because of this, TOFU uses user supplied input specifications in order to produce valid inputs to the SUT. They also use a different target guidance approach that is based on the distance between basic blocks in the CFG of the SUT.

7.5 Future Work

In this section we briefly motivate directions to move forward as a follow up to this thesis. We first discuss the need for a benchmarking suite and further tests in Section 7.5.1. Afterwards, we motivate approaches to improve performance and stability of the fuzzer and SUT in Sections 7.5.2 and 7.5.3. We then proceed with suggesting possible improvements to the mutation and seed selection components in Sections 7.5.4 and 7.5.5. Finally, we propose multiple approaches to further improve the state machine components in Sections 7.5.6 to 7.5.10.

7.5.1 Benchmarking

Comparing fuzzer effectiveness across different fuzzing approaches is difficult, because there is no de facto benchmarking strategy yet [Kle+18]. In this thesis we evaluate our approach against AFLNet as the baseline by using the achieved coverage as comparison metric. Hazimeh et al. [HHP20] argue, that this metric alongside with crash count or ground-truth bug counts is insufficient for use in fuzzer comparisons. They motivate the need of a benchmarking suite that contains a diverse range of SUTs with a number of known real-world bugs. However, their developed benchmarking suite Magma mostly considers stateless software.

A possible next step is to extend the Magma benchmarking suite to include more stateful software. Further, reevaluating the approach developed in this thesis against AFLNet and other fuzzers on the developed benchmark should provide additional insights.

7.5.2 Snapshots

As we have seen previously, using snapshots to rewind the program to a specific state, instead of restoring the state by re-sending the inputs, is able to increase fuzzer performance. However, the approach used in this thesis also is not portable and only works for programs that do not have external state in the form of for example configuration or database files. It sounds plausible to use virtualization, in order to alleviate the aforementioned problems, since creation and restoration of snapshots for virtual machines has already been implemented. Although this works, it is currently orders of magnitude slower than simply restarting the program from its initial state and restoring the state by sending all required inputs. The recent work by Schumilo et al. [Sch+21] tries to address this problem by developing methods to drastically increase the speed of virtual machine snapshot creation and restoration. Their approach is intended for hypervisor fuzzing but could be extended to stateful fuzzing.

7.5.3 Automatic Derandomisation

In the evaluation in Chapter 6, we saw that the stability of the SUTs influences the performance of the fuzzer. The stability is essentially a measure for how deterministic the SUT is. Derandomising targets often involves disabling the seeding of random generators and using deterministic clocks. Instead of relying on the user to derandomise the targets, a different approach could be to automatically derandomise the SUT. This could be done by fuzzing the program and identifiying sources of randomness, which can then be patched out by automatically recompiling the program or modifying the binary. However, this approach could be difficult to implement, since binary patching or source code modification is hard, when trying to preserve the functionality of the SUT. A different approach could be to mask out unstable regions in the coverage map. Although this should be easy to implement, it might not work depending on the SUT. If the randomness for example propagates through the program making most of it nondeterministic, this approach would not work.

Furthermore, multi threaded applications are inherently random if the kernel uses nondeterminism during scheduling. Such applications could be made deterministic by using a deterministic scheduling algorithm.

Finally, calls that perform some kind of I/O operation are also sources of randomness. Network operations for example depend on delays introduced by the network or kernel. A `recv` call might return different amounts of data, depending on how fast the data arrives and how it is reassembled. Also, file operations for example leave behind external state that needs to be cleaned up properly. Possible solutions to automatically deal with these problems include virtualization and mocking of syscalls. The virtualization however faces the same issues as

mentioned in Section 7.5.2. The preeny project [Sho20] already provides some methods to remove random seeding and redirect network sockets to stdin/stdout, and would be worth investigating for this purpose.

7.5.4 Directed fuzzing

Stateful fuzzing tries to separate the program into different states in order to concentrate fuzzing on those states that promise to yield the most results. In oder to concentrate the fuzzing on these states, AFLNet and the approach in this thesis limit the selected seeds to those that are able to reach this state. However, the mutation strategy that mutates the seeds is still random. Target oriented fuzzing [Böh+17; WLR20] or concolic execution [Asc+19; Ste+16; Yun+18] techniques could be applied in order to guide the fuzzer even further to code locations relevant to the selected state.

A different possible approach could be to utilize target oriented fuzzing in order to guide the fuzzer to code that promises to reveal more state information. These locations could either be extracted by using static analysis, or by using the feedback gained during fuzzing. Consider the case that multiple states have already been discovered. The fuzzer could then identify the code locations that led to the discovery of each state and focus on fuzzing those locations in order to find more states.

7.5.5 Seed Selection and Mutation Strategies

Böhme et al. [BMC20] describe Entropic, an approach using entropy as a measure of how much information is revealed about the SUT by individual seeds. They then use this entropy to weight the seeds, reasoning that seeds with high entropy promise to reveal more information about the SUT when fuzzed further, and achieve performance increases by doing so.

AFLNet and SNAPP currently use the standard AFL seed weighting for seed selection, and a completely random choice for the requests to use in sequence mutations. It seems plausible that adapting the entropy approach for seed selection in our context can boost performance as well. It should be investigated if and how the calculation of the entropy can incorporate information of the state machine. Furthermore, it could be worth investigating how entropy could be used for the request selection in the sequence mutator component. A possible idea is to adapt the ideas of Entropic to sequences of inputs, in order to find requests to insert into a sequence that increase the entropy of the sequence.

In addition, the ratio of sequence mutation to normal message mutation used in AFLNet and SNAPP (see Section 4.5), should be further analysed. Currently, AFLNet uses a fixed ratio and

SNAPP uses a weighted ratio depending on information gathered for each state. These values are currently chosen in an arbitrary way and need to be evaluated separately in the future.

Finally, the message mutation algorithm for SNAPP as well as AFLNet is currently adapted from AFL. Although these mutations proved to be effective, they do not consider dependencies between individual messages. Thus, a next step would be to devise techniques that are able to find such dependencies, allowing the fuzzer to direct its efforts towards fuzzing these values more effectively. Dynamic tainting as used by Mathis et al. [MGZ20] for example promises to be a good starting point for this.

7.5.6 State Machine Minimizing

The currently implemented approach only adds new states and as such the state machine only gets bigger. This could lead to problems when the amount of states gets too large, making it difficult to properly weight individual states. It is therefore worth investigating how the obtained state machine can be minimized. For example, if there is a sequence $\langle A,B,C,D\rangle$ with message M that leads to state S, the sequence could be analyzed in order to determine which messages are not necessary to obtain the behavior of M. If for example A can be omitted from the sequence without changing the behavior of M, the sequence could be shortened to $\langle B,C,D\rangle$. However, it should be noted that it is possible that A changed the state and information is lost by omitting it. It is possible that this state change would have been observed after fuzzing more messages with this sequence. As such, when to perform such a minimization step should also be looked into.

7.5.7 Message Behavior Sensitivity

In this thesis we used the exact coverage profile of a message as message behavior. However, this means that even the slightest change in coverage means that the behavior changes, and as such a new state is identified. It should be investigated if reducing the sensitivity of the message behavior could improve the performance of the fuzzer. A small change in execution does not necessarily correspond to a meaningful state change. Consider for example that two sequences simply causes a variable to be set to two different values. If the final message now causes a formatting function to be called, the formatting function might execute different paths for the two different values. Although this is interesting behavior, since new code was executed, it might not be necessary to categorize this as a new state.

A possible approach to reduce the message behavior sensitivity could be to group certain basic blocks together. Each basic block difference in such a group could then contribute to a kind of score of the group. The group would only be considered as different, if the score is

high enough, i.e. if enough basic blocks in the group differ. The work by Doupé et al. [Dou+12] uses a similar approach to ours. Instead of comparing the obtained responses of the SUT bit by bit, they group together the output obtained by the SUT using so-called link vectors obtained by parsing the response. Although this approach is very specific to their use case of fuzzing web applications, it can serve as starting point to reduce message behavior sensitivity.

7.5.8 Response Feedback

Currently, the approach of this thesis only uses the coverage feedback of the SUT in order to identify states. However, as mentioned in Section 7.4.1, blackbox approaches are also able to identify states. A next step would be to combine the two state identification techniques, either by keeping two separate state machine models, or by combining the identification approaches into one. Still, regardless of how the approaches are combined, since one of our goals was to achieve protocol independence, the black box approach has to be protocol independent.

7.5.9 Automatic Injection

The current approach described in Chapter 4 requires injecting a function into the mainloop of the program. Since one of the goals of the automatic state identification was to reduce manual work required, a next step would be to automate the injection. A possible approach could be to use the CFG of the SUT in order to identify the main loop.

Also, a different concept to modifying the SUT by injecting code for synchronisation could be to use the ptrace API [Ker20]. The ptrace API is used by debuggers to for example set breakpoints and get notified once they are hit. Using this API would have the benefit that the SUT does not have to be recompiled in order to enable the synchronisation. This also makes it possible for the fuzzer to adaptively change the synchronisation point if necessary.

7.5.10 Memory Feedback

The current approach of identifying the states does not cover states that are exclusively represented in memory. That is, if the program executes the same code twice, the state can still be different. For example, consider two different inputs. If they are just passed on into memory, the code paths are the same, but memory content differs. Although the SUT executes the same code for both inputs, the content saved in memory can produce different execution paths later on. Coverage will not be a helpful metric in this case. The fuzzer would only save one of the inputs to the seed corpus and discard the second one, even though the input might have been more interesting later on, when the content of the memory is read.

Although the IJON paper by Aschermann et al. [Asc+20] tackles this problem by using annotations to enable feedback for situations like this, their approach still requires inserting the annotations manually in the right locations. It should be investigated how to identify such locations, and how to insert the correct annotations. In order to identify locations where annotations can be placed it might be possible to use the number of accesses to specific memory addresses. The intuition behind this being, that many accesses to the same memory address corresponds to a location where state is stored.

7.6 Overall Findings

The performed evaluation shows that the new automatic state identification approach developed in this thesis and prototypically implemented as SNAPP achieves similar or better results compared to AFLNet. However, in contrast to AFLNet we are able to achieve these results without the need for manual specification. Furthermore, our approach seems to be able to test deep state dependencies better than AFLNet when given a large initial corpus. This is the first step to a fully automated stateful greybox fuzzer, with the second step consisting of eliminating the need for a manually specified request sequence parser. We can also conclude that the selection of the initial corpus plays a key role if messages are not generated according to a generator specification. The importance of the initial corpus selection could also be reduced when eliminating the manual request sequence parser, if a generator is used to automatically generate an initial seed according to an automatically extracted specification. Overall, we can conclude that SNAPP provides an improvement over AFLNet, and a good starting point to even further increase automation and efficiency for stateful greybox fuzzing.

8 Conclusion

Fuzzing more types of software and providing better usability, e.g. automating more steps in the fuzzing process, is identified by Böhme et al. [BCR20] as one of many current challenges. They consider especially fuzzing stateful software as open research. The objective of this thesis was thus to design and evaluate new methods for efficient fuzzing of stateful software. In order to achieve this objective, we analyzed the existing approaches, and identified AFLNet as the only stateful fuzzer that currently follows a greybox approach. We saw that AFLNet by Pham et al. [PBR20b] is able to significantly outperform classic fuzzers like AFL and also blackbox approaches like boofuzz. However, we identified that AFLNet needs a manually specified extraction function to identify states, reducing the adaptation rate of the fuzzer, because of the manual work required when adapting to new SUTs. In order to alleviate this, we conceived a new algorithm that automatically infers a state machine during fuzzing of the SUT. We also described a synchronisation mechanism required to make the program more deterministic, such that the automatic state identification is able to function correctly. In order to evaluate our approach we implemented the algorithms using AFLPlusPlus as basis, and called our new fuzzer SNAPP. We then performed an evaluation on the three open source targets open62541, lightftp, and live555, of which the latter two were also tested in the original AFLNet paper [PBR20b]. The evaluation compared the fuzzers' achieved coverages over the course of 48 hours each, and for one evaluation run on open62541 over the course of 7 days. The achieved results indicate that SNAPP performs at least as well as AFLNet in most cases, and in some cases even performs better than AFLNet. However, in contrast to AFLNet, SNAPP does not need manual specification in order to infer a state machine, while still achieving similar or better performance. Especially for the 7-day evaluation done on a large initial corpus, SNAPP outperformed AFLNet by 3.22% in a statistically significant manner when comparing the achieved final coverage. We also saw that the initial corpus choice makes a significant difference, resulting in a 22.95% increase for SNAPP when comparing the final achieved coverage of the small and large initial corpora. The evaluation further showed, that the performance of SNAPP can be further increased by using a primitive state snapshotting mechanism.

Overall, this thesis showed that coverage feedback can successfully be employed to automatically identify states in order to achieve similar or better coverage than the state of the art greybox stateful fuzzer AFLNet, without needing to manually specify an extraction function for the states. We also showed that the initial corpus choice is important, and motivated a plethora of directions to move forward in.

Bibliography

[AGP19] V. Atlidakis, P. Godefroid, and M. Polishchuk. "RESTler: Stateful REST API Fuzzing." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 748–758.

[Aiz+16] Mike Aizatsky et al. *Announcing OSS-Fuzz: Continuous fuzzing for open source software*. 2016. URL: https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html (visited on 12/04/2020).

[Asc+19] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. 2019.

[Asc+20] C. Aschermann et al. "IJON: Exploring Deep State Spaces via Fuzzing." In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 893–908. URL: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00050.

[BCR20] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and Reflections." In: *IEEE Software* PP (Aug. 2020).

[BMC20] Marcel Böhme, Valentin Manes, and Sang Kil Cha. "Boosting Fuzzer Efficiency: An Information Theoretic Perspective." In: (2020).

[Böh+17] Marcel Böhme et al. "Directed Greybox Fuzzing." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. URL: https://doi.org/10.1145/3133956.3134020.

[Cab+07] Juan Caballero et al. "Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis." In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 317–329. URL: https://doi.org/10.1145/1315245.1315286.

[ClV19] Yurong Chen, Tian lan, and Guru Venkataramani. "Exploring Effective Fuzzing Strategies to Analyze Communication Protocols." In: *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. FEAST'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 17–23. URL: https://doi.org/10.1145/3338502.3359762.

[Com18] International Electrotechnical Commission. *IEC 62443 Security for Industrial Automation and Control Systems Standard*. 2018.

[Cui+08] Weidong Cui et al. "Tupni: Automatic Reverse Engineering of Input Formats." In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 391–402. URL: https://doi.org/10.1145/1455770.1455820.

[CVE13] *CVE-2014-0160 Heartbleed bug*. Dezember 2013. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160 (visited on 12/20/2020).

[CVE14] *CVE-2014-6271 Shellshock*. Sept. 2014. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-6271 (visited on 12/20/2020).

[Don+20] Zhen Dong et al. "Time-travel Testing of Android Apps." In: (May 2020).

[Dou+12] Adam Doupé et al. "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner." In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 523–538. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[FC18] Rong Fan and Yaoyao Chang. "Machine Learning for Black-Box Fuzzing of Network Protocols." In: *Information and Communications Security*. Ed. by Sihan Qing et al. Cham: Springer International Publishing, 2018, pp. 621–632.

[Fit+20] Paul Fiterau-Brostean et al. "Analysis of DTLS Implementations Using Protocol State Fuzzing." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. "W32. stuxnet dossier." In: *White paper, Symantec Corp., Security Response* 5.6 (2011), p. 29.

[Fou17a] OPC Foundation. *OPC Unified Architecture Specification Part 1: Overview and Concepts*. Version 1.04. OPC Foundation. Nov. 2017. URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/.

[Fou17b] OPC Foundation. *OPC Unified Architecture Specification Part 4: Services*. Version 1.04. OPC Foundation. Nov. 2017. URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-4-services/.

[Fou17c] OPC Foundation. *OPC Unified Architecture Specification Part 6: Mappings*. Version 1.04. OPC Foundation. Nov. 2017. URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-6-mappings/.

[Gas+15] Hugo Gascon et al. "Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols." In: *Security and Privacy in Communication Networks*. Ed. by Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran. Cham: Springer International Publishing, 2015, pp. 330–347.

[Gau20] Roman Gaufman. *Live555 Git Mirror*. 2020. URL: https://github.com/rgaufman/live555 (visited on 12/04/2020).

[Goo20a] Dan Goodin. *Microsoft president calls SolarWinds hack an act of recklessness*. Dec. 18, 2020. URL: https://arstechnica.com/information-technology/2020/12/only-an-elite-few-solarwinds-hack-victims-received-follow-on-attacks/?comments=1 (visited on 12/20/2020).

[Goo20b] Dan Goodin. *Russian hackers hit US government using widespread supply chain attack*. Dec. 18, 2020. URL: https://arstechnica.com/information-technology/2020/12/russian-hackers-hit-us-government-using-widespread-supply-chain-attack/?comments=1 (visited on 12/20/2020).

[Goo20c] Google. *Honggfuzz*. 2020. URL: https://honggfuzz.dev/ (visited on 12/08/2020).

[HB14] *The Heartbleed Bug*. Apr. 2014. URL: http://heartbleed.com/ (visited on 12/20/2020).

[Heu+20] Marc Heuse et al. *American Fuzzy Lop plus plus (afl++)*. 2020. URL: https://github.com/AFLplusplus/AFLplusplus (visited on 12/22/2020).

[HHP20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma." In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (Nov. 2020), pp. 1–29. URL: http://dx.doi.org/10.1145/3428334.

[Iat+20] Chris-Paul Iatrou et al. *open62541 Git Repository*. 2020. URL: https://github.com/hfiref0x/LightFTP (visited on 12/04/2020).

[Ker20] Michael Kerrisk. *ptrace Linux Manual Page*. 2020. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 12/17/2020).

[Kle+18] George Klees et al. "Evaluating Fuzz Testing." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. URL: https://doi.org/10.1145/3243734.3243804.

[laf16] lafintel. *Circumventing Fuzzing Roadblocks with Compiler Transformations*. Aug. 2016. URL: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/ (visited on 06/24/2020).

[Li+17] Yuekang Li et al. "Steelix: Program-State Based Binary Fuzzing." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 627–637. URL: https://doi.org/10.1145/3106237.3106295.

[LLV20] LLVM-Project. *libFuzzer*. Nov. 2020. URL: https://llvm.org/docs/LibFuzzer.html (visited on 11/17/2020).

[Ma+16] R. Ma et al. "Test data generation for stateful network protocol fuzzing using a rule-based state machine." In: *Tsinghua Science and Technology* 21.3 (2016), pp. 352–360.

[Man+19] V. J. M. Manès et al. "The Art, Science, and Engineering of Fuzzing: A Survey." In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1.

[MGZ20] Björn Mathis, Rahul Gopinath, and Andreas Zeller. "Learning Input Tokens for Effective Fuzzing." In: *ISSTA - ACM SIGSOFT International Symposium on Software Testing and Analysis*. July 2020, pp. 1–11. URL: https://publications.cispa.saarland/3135/.

[MW47] H. B. Mann and D. R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other." In: *Ann. Math. Statist.* 18.1 (Mar. 1947), pp. 50–60. URL: https://doi.org/10.1214/aoms/1177730491.

[Myt+09] Todd Mytkowicz et al. "Producing Wrong Data without Doing Anything Obviously Wrong!" In: *SIGPLAN Not.* 44.3 (Mar. 2009), pp. 265–276. URL: https://doi.org/10.1145/1508284.1508275.

[MZH20] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. *The Relevance of Classic Fuzz Testing: Have We Solved This One?* 2020. arXiv: 2008.06537 [cs.SE].

[PBR20a] Van - Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. *AFLNet Github Repository*. 2020. URL: https://github.com/aflnet/aflnet (visited on 12/09/2020).

[PBR20b] 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.

[Per20] Joshua Pereyda. *boofuzz*. 2020. URL: https://boofuzz.readthedocs.io/en/stable/ (visited on 12/08/2020).

[PP16] Jibesh Patra and Michael Pradel. "Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data." In: (2016).

[Pro20] LightFTP Project. *LightFTP Git Repository*. 2020. URL: https://github.com/hfiref0x/LightFTP (visited on 12/04/2020).

[RP15] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations." In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[Sal+20] C. Salls et al. "Exploring Abstraction Functions in Fuzzing." In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020, pp. 1–9.

[Sch+21] Sergej Schumilo et al. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types." In: *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo.

[Sel14] Larry Seltzer. *Shellshock makes Heartbleed look insignificant*. Sept. 29, 2014. URL: https://www.zdnet.com/article/shellshock-makes-heartbleed-look-insignificant/ (visited on 12/20/2020).

[Sho+16] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *IEEE Symposium on Security and Privacy*. 2016.

[Sho20] Yan Shoshitaishvili. *Preeny*. 2020. URL: https://github.com/zardus/preeny (visited on 12/18/2020).

[Ste+16] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[Tak+18] A. Takanen et al. 2018.

[TC07] Linda Torczon and Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[Tec20] Peach Tech. *Peach Fuzzer Community Edition*. Dec. 16, 2020. URL: https://www.peach.tech/resources/peachcommunity/.

[Wan+20] Haijun Wang et al. "Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 999–1010. URL: https://doi.org/10.1145/3377811.3380386.

[Whe17] David A. Wheeler. *How to Prevent the next Heartbleed*. Jan. 2017. URL: https://www.dwheeler.com/essays/heartbleed.html (visited on 12/20/2020).

[WLR20] Zi Wang, Ben Liblit, and Thomas Reps. "TOFU: Target-Orienter FUzzer." In: *arXiv preprint arXiv:2004.14375* (2020).

[Yun+18] Insu Yun et al. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing." In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, pp. 745–761.

[Zal+20] Michal Zalewski et al. *American Fuzzy Lop*. July 2020. URL: https://github.com/google/AFL (visited on 08/18/2020).

List of Tables

List of Figures

Definitions and Theorems

List of Algorithms

Listings

Glossary

CFG control flow graph. 12, 14, 15, 89, 93, 105

FTP File Transfer Protocol. 19, 22

Fuzzer A Fuzzer is a tool that more or less randomly mutates inputs in order to trigger bugs in a target executable.

OPC UA OPC Unified Architecture (OPC UA) is a machine to machine communication protocol for industrial automation developed by the OPC Foundation. 4–7, 13, 20, 33, 51–54, 58, 103, 105, 111

RTSP Real Time Streaming Protocol. 19, 22, 85

SMT satisfiability modulo theories. 10

SUT system under test. 1–3, 5, 7–11, 16–22, 25, 27–29, 31, 33–38, 40, 45, 46, 48, 50, 51, 53–62, 74, 77, 78, 80, 81, 85–91, 93, 95, 103, 105

TCP Transmission Control Protocol. 6, 56, 75