Bachelor thesis
Computing Science

Radboud University

Approaches to stateful fuzzing of the IRC protocol

Author:
Patrick Lodeweegs
S1027584

First supervisor/assessor:
Dr. ir. E. Poll
erikpoll@cs.ru.nl

Second assessor:
Drs. ing. P. Van Aubel
pol.vanaubel@cs.ru.nl

June 17, 2021

Abstract

To find implementation issues in software, automated testing can be applied. An effective way to find security vulnerabilities in a system under test (SUT) is fuzzing.

Fuzzing in a simple form sends a larger number of randomly generated inputs to a SUT and checks unexpected behaviour. For programs implementing a stateless protocol, this can already be an effective strategy. However, for stateful protocols, more advanced methods are required to reach deeper states. For instance, a deeper state is only reached after completion of a particular sequence of messages.

To optimise fuzzing of stateful protocols, four approaches have been evaluated in this thesis. These approaches are all based on the stateful greybox fuzzer AFLNet with the IRC protocol as a case study. This fuzzer has been extended to support:

1. the IRC protocol as baseline;
2. a dictionary for the IRC protocol;
3. state-aware fuzzing based on IRC responses;
4. checkpointing for any protocol supporting state-aware mode.

For approaches one, three, and four, new functionality has been implemented. The second approach, using a dictionary, is directly supported by AFLNet. For many protocols dictionaries are supplied, however, for the IRC protocol no dictionary was previously available. Therefore, a new dictionary has been constructed. The principles of these four approaches can be used for most stateful protocols. The approaches are empirically compared on path coverage, maximum depth and crashes found, using a case study on the Internet Relay Server ngIRCd.

Significantly more paths have been found when using a dictionary compared to the baseline. State-aware fuzzing does not improve the total paths found and maximum depth significantly. Combining these approaches resulted in a decrease in coverage and depth compared to using only a dictionary.

While using the four approaches, no new bugs were found in the SUT. To estimate whether there are no bugs in the SUT or whether the scope of the experiments was too limited, a number of synthetic bugs has been introduced in the SUT. Several of these bugs have been found by the fuzzer. This suggests the scope of the conducted experiments is sufficient.

Contents

Chapter 1

Introduction

Fuzzing is an approach to automate software testing by sending several test inputs to a target system. Unexpected behaviour of a system under test (SUT), such as a crash or hang, is monitored by the fuzzer during the procedure. This can be used to identify security vulnerabilities in an implementation that May remain unnoticed by manual code inspection [1].

Often fuzzing is directed towards stateless protocols. In this context, stateless protocols are programs where the program input cannot be divided into a sequence of independent messages. This implies that all program states can be reached with a single input message. An example of a stateless SUT is a PDF reader that can be fuzzed by creating several input PDF files.

Stateful protocols require input message sequences to reach deeper states. For instance, a deeper state might be only reachable after completing a handshake sequence successfully [2]. If a fuzzer does not consider input as a sequence, this creates a risk of directing most fuzzing efforts to some initial states of a SUT. As a result, stateful protocol implementations are often assumed to be more complicated to fuzz effectively.

This thesis evaluates the improvement of four measures to optimise fuzzing in a stateful protocol implementation. Network protocols are a widely used example of stateful protocols. A case study is conducted to fuzz a specific network protocol. In this case study, the SUT is an implementation [3] of the IRC protocol [4].

Fuzz testing in this thesis is focused on client-to-server communication. Background information about several fuzzing techniques is given in chapter 2. To test several measures to optimise fuzzing of stateful protocols, a multistep approach, is applied. As initial step, several fuzzers are assessed. Most widely used fuzzers do not support the same protocols. Therefore, instead of evaluating fuzzers based on specific protocol support, fuzzers are evaluated on the support of stateful fuzzing. In chapter 3, a number of fuzzing frameworks is evaluated.

The IRC protocol is a standardised protocol for text messaging. Origi-

nally specified in 1993, this protocol is updated with more recent RFCs and with implementation specific modifications. Both client-to-server as server-to-server communication is possible. Server-to-server communication allows several servers to connect and form a network. More information regarding the IRC protocol is provided in chapter 4.

Base on the evaluation of several fuzzers, AFLNet has been chosen as basis for the experiments in chapter 5. To compare several approaches, a baseline is created. For this baseline, AFLNet is extended to support fuzzing of the IRC protocol by implementing a request extraction function for IRC. Additionally, the IRC server is prepared for fuzz testing. More details regarding testing the first approach are described in section 5.3.

Further experiments are used to test approaches to grammar based fuzzing with a dictionary (section 5.4), state-aware fuzzing (section 5.5) and checkpointing (section 5.6):

- Creating a dictionary enables the fuzzer to use high level information regarding the protocol, which can help reaching deeper states.
- state-aware fuzzing tries to determine the state of the SUT. This could be used to aid the fuzzer to cover all known states of the SUT.
- Stateful protocols require a sequence of input messages to reach deeper states. Part of this sequence forms a prefix that has to be repeated each time to fuzz a deep target state. By using checkpointing, this prefix can be saved and restored the next time that a specific state is fuzzed.

These approaches are compared in experiments based on the evaluation measures defined in section 5.2.1. Additional experiments have been designed to determine (i) the sufficiency of the scope of the previous experiments and (ii) the impact of the initial seed. In case fuzz testing does not find any vulnerabilities, this can imply that either a SUT has no detectable vulnerabilities or the time budget of the fuzzer was too low. To distinguish between the first and last two categories, synthetic vulnerabilities have been injected as described in section 5.8.

Alternative approaches regarding stateful fuzzing are identified in this thesis. These are summarised as recommendations for further research in chapter 6. Finally, in chapter 7 the overall conclusions of this thesis are discussed.

Chapter 2

Background on fuzzing

In this chapter background information regarding stateful fuzzing techniques is presented. Fuzzing is often used in an automated software testing setup. Depending on the SUT, there are several approaches possible. In general, three high-level approaches to fuzz testing are distinguished [2]. A high level overview of blackbox and whitebox fuzz testing is given in section 2.1. More details regarding greybox fuzzing, a specific approach to fuzz testing, are given in section 2.2. Greybox fuzzing is applied in the experimental settings introduced in this thesis in chapter 5. Some often used techniques to mutate messages are explained in section 2.3. To measure performance of fuzzing, some feedback metrics are introduced in section 2.4.

2.1 White- and blackbox fuzzing

Blackbox fuzzing is an approach where input is generated without implementation knowledge of the SUT. This approach is successfully used to test for instance compilers and database systems [5]. In a simple form blackbox fuzzing executes random test cases. An often implemented variant attempts to minimise the similarity of random tests over the input space. This variant is based on the concept that similar test cases will often test the same part of a code base and therefore yield similar results. To promote different results, an equal division in partitions is an option to ensure that all test cases are overall the least similar. Proportional sampling from partitions is proven to give better results than strictly random testing in some cases [6]. In practice, sampling from the specified input space results in a focus mostly on the specified behaviour. However, some vulnerabilities are only triggered by undefined behaviour and may be missed by this approach.

On the other side of the spectrum, there is whitebox fuzzing. This approach requires full access to the source code of the SUT. Often program analysis techniques such as symbolic execution and constraint programming are applied [7, 8]. These techniques enable whitebox fuzzers to ideally

execute a new path on each execution. In larger programs there often too much paths to do an exhaustive search. With a low time budget, this results in whitebox fuzzing not outperforming blackbox fuzzing for programs with a complex input [9].

2.2 Greybox fuzzing

A compromise between blackbox and whitebox fuzzing approaches is greybox fuzzing. Greybox fuzzing incorporates some feedback from the SUT, often by either automatically or manually instrumenting key locations such as branch points. Compared to black box fuzzing, this method often results in more program coverage. However, especially manual defined feedback, can introduce bias to specific program states.

Since greybox fuzzing makes use of feedback from the SUT, several forms of feedback can be applied. Based on received feedback from the SUT, other techniques can be applied to increase effectivity of a fuzzer. Feedback from the SUT is often used to guide the fuzzing process. Depending on the type of feedback, a metric can be generated automatically or requires manual annotation. Metrics commonly used and introduced in section 2.4 are code coverage, program state coverage, branch coverage, number of unique crashes and number of unique bugs [10, 11]. Some metrics such as unique crashes and unique bugs can also be used as metric for white- and blackbox fuzzing.

2.3 Input generation strategies

The first known fuzzer has been used in 1990 to test a corpus of 90 programs. This fuzzer used randomly constructed characters as input for the tested systems. This method found vulnerabilities in more than 24% of the systems [12].

Since most random input is expected to be invalid, a SUT often rejects an input in an early state. Therefore, a completely random fuzzer might not reach deeper states of a SUT. A popular approach to automatically generate input with a higher validity probability is using random mutations of an input sequence [13]. One of the first popular mutation based fuzzers is AFL [14]. AFLNet, a derivative of AFL is described in section 3.1. Both AFL and AFLNet mutate input messages that uncover new transitions between states of the SUT. These inputs are used as seed for further mutations [1].

A different, but not mutually exclusive approach to input generation is applying a grammar. A grammar can be used to generate syntactically valid input messages. Especially, if a SUT uses a complex input this could help to reach deeper states. However, creating a complete grammar of a protocol might be challenging depending on the protocol complexity.

[1] https://afl-1.readthedocs.io/en/latest/about_afl.html#how-afl-works

Additionally, certain rules in a grammar could have a higher probability to be used compared to other rules [15]. Optimally assigning such probabilities is a non-trivial problem. An approach to get reasonable initial values is by measuring frequencies of rules occurring in a larger sample. Promising results are also achieved by using an evolutionary process to increase the probability of generating interesting rules at runtime [16].

These input generation strategies can be applied not only on stateful fuzzing but also on stateless fuzzing. However, some additional considerations are useful depending on if the SUT is stateful. For instance fuzzing a stateless image library requires only a single input for each iteration. Therefore, when using mutations to generate new inputs, a mutation could in principle occur in the complete input space. On the other hand, when fuzzing a stateful protocol it can be useful to mutate only a specific part of the input. This input forms a sequence where later parts of the sequence depend on previous parts. Therefore, mutations in the entire input sequence could result in fuzzing only initial states.

2.4 Feedback metrics

Performance metrics can be used to evaluate the performance of a fuzzer on a SUT. Additionally, greybox fuzzers can use certain metrics as feedback during fuzzing.

Code coverage is a relatively simple metric where the number of lines of the SUT executed during fuzzing is counted. This can be compared without manual validation. Therefore, code coverage can be used as an objective metric that correlates with the number of bugs [17]. Intuitively, full code coverage is a requirement to find all bugs in a SUT. However, the inverse that all bugs in a SUT are found if full code coverage is achieved does not hold true.

Program state coverage forms an abstraction of code coverage. Parts of a SUT are grouped together based on, for instance, variable instantiation or execution path to form a state. State coverage can yield more insight in the actual functionality of a program, but requires a state machine model. Such a model can either be generated or provided by specification. Generation or learning of a state model can be done in advance or while fuzzing. To assist in this process there are tools such as LearnLib that implement several automata learning algorithms [18]. Sometimes a state machine is provided with the SUT or protocol specification. More details regarding a state machine for the IRC protocol are provided in section 5.5.

Branch coverage can be used to keep track of how many different execution paths are found by the fuzzer. Similary to state coverage, branch coverage can be regards as an abstraction of code coverage. Execution paths

can be used to gain understanding in the flow of a SUT, which aids in manually analysing bugs.

The number of unique crashes can be approximated during the fuzzing of a SUT. A new crash can be compared with all previous occurred crashes. In case a crash is not equivalent to previous crashes, the crash is seen as a unique crash. Different methods to define equivalence could introduce bias. Bias could result in under- or overcounting the number of crashes. An example that is expected to result in overcounting crashes is defining equivalence if several crashes have an identical checksum of the call stack. Undercounting could occur if equivalence is based on the instruction address where the crash occurs[2].

The number of bugs found in SUT as performance metric, aligns well with the intention of a fuzzer to find vulnerabilities. However, to determine a unique bug from a set of crashes is often a manual process. A single bug might result in several crashes in different parts of the SUT, depending on the input message sequence. This makes this metric more suitable for an overall performance evaluation than as a heuristic during fuzzing.

[2]https://github.com/aflnet/aflnet/blob/master/docs/technical_details.txt

Chapter 3

Fuzzing frameworks

In this chapter several fuzzing frameworks based on AFL are described. AFL is a popular foundation for domain specific fuzzers such as AFLNet, AFLNwe and AFLSmart. Based on the overall evaluation in section 3.4, the greybox fuzzer AFLNet is chosen as base program for experiments in chapter 5 [11]. More details regarding AFLNet are provided in section 3.1. Boofuzz, another greybox fuzzer, is evaluated in section 3.2.

Greybox fuzzing frameworks are often targeted at programs implementing stateless protocols as described in chapter 2. In this context a program implementing a stateless protocol is defined as a protocol where a set of single inputs can cover the entire SUT. This kind of programs is further referred to as stateless programs. Programs implementing stateful protocols require a set of input sequences to cover the SUT instead. Such programs are referred to as stateful programs. To fuzz stateful programs with a fuzzer intended for stateless programs could reduce performance (see section 2.3). Blackbox fuzzing is currently often the preferred fuzzing approach for stateful programs [11]. Peach, a blackbox fuzzing framework is analysed in section 3.3.

3.1 Greybox fuzzing with AFLNet

AFLNet is an extended version of AFL which implements strategies to fuzz stateful programs in addition to most features of AFL [11, 14]. Most properties of AFL also hold for its derivatives such as AFLNet, AFLSmart and AFLNwe.

A difference between AFLNet compared to AFL is that AFL is intended to fuzz stateless programs. To enable stateful fuzzing, AFLNet implements both sending messages to and receiving messages from the SUT. AFLNet requires an initial seed. This seed consists of a set of messages. These messages are recorded packets of regular or simulated traffic to the SUT in the PCAP Capture File Format [19]. Such recordings can be made using

for instance Wireshark[1] or tcpdump[2]. Since, only messages to the SUT are used by the fuzzer as initial input, responses of the SUT can be discarded.

AFL applies a mutation based algorithm to create input message sequences. AFL supports several mutation strategies. For instance mutations can consist of replacement, duplication, insertion and deletion of a message or a block of bytes in a message. To fuzz stateful programs, AFLNet mutates a message sequence by first dividing it in three parts. The first part of the sequence is considered to be a prefix. This part is required to reach a certain state and will therefore not be mutated. The second part consists of all messages where the SUT will remain in the same target state. In this part a mutation will occur. The last part is considered irrelevant for the current target state. This part will, similar to the first part, not be mutated. For the next fuzzing iteration all three parts are combined to a single sequence again.

During fuzzing, both AFL and AFLNet make use of instrumentation of the SUT to guide the fuzzer. Modified compiler variants for automatic instrumentation of C and C++ are provided. Projects to instrument other programming languages are also available. In the experimental setup compile time instrumentation is used as baseline in section 5.3. In case only a binary version of a SUT is available, QEMU is integrated for instrumentation without additional preparation [20].

3.1.1 AFLNwe and AFLSmart

Like AFLNet, AFLNwe and AFLSmart are both extensions to AFL. Practically, AFLNet can be seen as an extension of AFLNwe and AFLSMart too. AFLNwe is a minimally extended version of AFL that has only extensions added to enable fuzzing network protocols using sockets [11]. This potentially could make AFLNwe simpler to extend than AFLNet.

In AFL and AFLNet a protocol can be fuzzed with the help of a dictionary. Such a dictionary forms a basic grammar that describes the protocol. This does not solve mutations on an abstract protocol level instead of a message level directly. AFLSmart, which is mostly developed by the same team of researchers as AFLNet, attempts to mitigate this issue by applying mutation operators that work on the virtual structure of the file instead. AFLSmart introduced interesting concepts regarding fuzzing of a protocol such as region-based mutation. Region-based mutations allow mutating multiple symbols at once, which substantially improves branch coverage [21]. This functionality is also integrated in AFLNet.

[1] https://www.wireshark.org/
[2] http://www.tcpdump.org/

3.2 Greybox fuzzing with Boofuzz

Boofuzz[3] is a popular fuzzing framework for stateful SUTs [2]. Boofuzz uses protocol specifications to generate message sequences. Additional instrumentation and monitoring of the SUT allows analysing and categorising faults. Additionally, a form of checkpointing is built which enables reverting to a known non-crashing state. Boofuzz is a further developed version of Sulley[4] and among others enables easier extension. In literature there is no clear consensus whether to classify Boofuzz as whitebox or blackbox fuzzer [2, 22]. By the definition used in this thesis defined in chapter 2, Boofuzz is classified as greybox fuzzer.

3.3 Blackbox fuzzing with Peach

Blackbox fuzzing uses at most protocol specifications to fuzz a SUT. Since only the input and output of the SUT is known, a blackbox fuzzer cannot use instrumentation to increase performance. Peach is a popular blackbox fuzzing framework[5] first developed in 2004[6][23]. Like Boofuzz, Peach is a generation based fuzzing framework. This allows generating new input messages that preserve integrity constraints. This approach helps to reach deeper states of a SUT [7].

3.4 Comparison of fuzzing frameworks

Based on a comparison with some other popular fuzzers described in this chapter, AFLNet has been chosen as a base for the experiments in this thesis. AFLNet is a maintained project with an open source licence (Apache-2.0). However the code base itself is not extensively documented. Additionally, parts of the code base appear to have duplicate code.

AFLNet is based on AFL. AFL is one of the most prominent greybox fuzzers and is tested in practise since 2014[7]. Besides a quite extensive first party documentation, there are many third party articles available for AFL that are also mostly applicable to AFLNet.

Unlike AFL and AFLSmart, AFLNet has support for stateful network protocols. AFLNwe and Boofuzz also support stateful protocols. However, According to the benchmarks conducted by introducing AFLNet there is a statistically significant difference in most test cases in favour of AFLNet compared to AFLNwe and Boofuzz.

[3] https://github.com/jtpereyda/boofuzz

[4] https://github.com/OpenRCE/sulley

[5] https://github.com/asudhak/peachfuzz-code

[6] https://web.archive.org/web/20201001072717/https://community.peach-fuzzer.com/WhatIsPeach.html

[7] https://lcamtuf.coredump.cx/afl/releases

AFLNet does not support fuzzing with a protocol grammar. Advantages of using a grammar to fuzz complex protocols are described in section 2.3. However, AFLnet tries to reconstruct a grammar at run time based on responses of the SUT. Optionally, this grammar reconstruction can be assisted by creating a dictionary for a SUT[8]. This enables AFLNet to create with relatively high probability valid input messages, without crafting a detailed grammar for each target protocol. Such a dictionary is constructed for the IRC protocol in section 5.4.

[8]https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

Chapter 4

Background on IRC

In this chapter background information regarding the IRC protocol and a server implementation is presented. A SUT implementing the IRC protocol is used in the case study. The case study is presented in chapter 5.

The IRC protocol is documented in several Request for Comments (RFCs). The original specifications were drafted in RFC 1459 from 1993. Further revisions were made in RFCs 2810, 2811, 2812, 2813 and 7194. RFC 1459 is mostly superseded by RFCs 2810 till 2813. RFC 7194 only suggests a standardised approach to transport layer security for client-to-server (C2S) communication. In this research the main focus will be on the application level IRC protocol as described in RFCs 2810 till 2813.

- RFC 2810 Architecture [4]: This RFC describes the architecture of the most recent IRC protocol specification with a high level overview of its different sub components.
- RFC 2811 Channel management [24]: This RFC describes channel management by IRC servers. Channels can have a namespace and several modes. These modes can be used as permission management.
- RFC 2812 Client Protocol [25]: This RFC specifies the C2S communication protocol.
- RFC 2813 Server Protocol [26]: Due to the federated nature of IRC, server-to-server (S2S) communication occurs not only as C2S but also between servers. This RFC describes the protocol between servers.

Both the C2S and S2S protocol are stateful. The C2S protocol is widely used with several third party clients such as Quassel IRC[1] and Weechat[2]. These IRC clients are typically used by an end user and expose the chat functionality of the IRC protocol. Often a single IRC server is used for several

[1] https://github.com/quassel/quassel
[2] https://github.com/weechat/weechat

clients. Besides end user clients, the IRC protocol additionally defines service clients. Service clients are often automated to provide some additional service to an end user. These clients should have limited access to IRC protocol functionality. In this thesis fuzz testing is mostly directed to the end user client communication with the server.

In section 4.1 more details of a message between the client and server are given. Some notable protocol extensions are highlighted in section 4.2. On the choice of an IRC server implementation as SUT more information is given in section 4.3.

4.1 RFC 2812: Client protocol

RFC 2812 is the latest RFC that specifies the C2S part of the IRC protocol. Based on this RFC several requirements for a valid IRC message can be constructed.

According to section 2.3 of RRC 2812, any valid request of the client should get a response. There is no maximum response time specified. Anyway, when fuzzing on a local system long timeouts are not expected.

An IRC response consists of at most three parts. Each part is separated with a space (0×20). The first part is an optional prefix. This prefix is indicated with a special character at the start of a message. In the RFC, there is a mistake regarding the prefix presence indication symbol. In practice, this is done with a colon character ($0 \times 3a$) instead of a semicolon character ($0 \times 3b$). This error is corrected in errata 384[3]. The prefix is in principle only used from server responses to the client to indicate the origin of a message. The second part of a response message contains a valid IRC command or a three-digit numerical representation of a command. A full list of numeric commands is given in RFC 2812 section 5.1.

The final part of a message consists of command parameters. Such a parameter is for instance used by a server response message to acknowledge a request of a client. A complete message consists of at most 512 characters. This includes then optional initial prefix and the message termination character `CR-LF` ($0 \times 0d0a$).

To register on an IRC server there are several message sequences possible. The recommended registration sequence is shown in fig. 4.1. The Password message is optional depending on the server configuration. Also, clients are allowed to first send a User message before the Nick message. Specifying such variations results in a more complex code base, which increases the risk of vulnerabilities.

[3] https://www.rfc-editor.org/errata/eid384

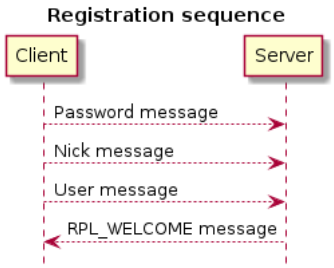


Figure 4.1: The recommended sequence for registration on an IRC server according to RFC 2812. This diagram is generated with PlantUML [27]

4.2 IRC protocol extensions

The most recent major updates of RFCs regarding the IRC protocol are made in the year 2000. To adapt to new requirements, there is an ongoing effort to document a next iteration of the IRC C2S protocol as IRCv3. This specification effort explicitly excludes S2S connections. Currently, some IRC servers support a subset of the proposed IRCv3 extensions. For instance the ngIRCd server described in section 4.3 partially implements IRCv3 proposals. Additionally, this server implements some extensions in the form of the "IRC+" protocol. This mode is active by default, however it is possible to enable an RFC only mode[4].

4.3 IRC server implementations

There are several implementations of IRC servers. For greybox fuzzing it is preferred that the source code of an implementation is available. An available source code gives several advantages for fuzz testing. Automated instrumentation, if done at compile time, results in less performance overhead. Additional optimisations, such as disabling non-essential parts of the program, can be applied to improve fuzzing performance. Moreover, by disabling non-essential parts, specific bugs can be easier identified, in case potential vulnerabilities are found.

Lower level languages such as C and C++ have more potential for memory related issues. Especially when using stack protectors and sanitizers, this kind of issues can be identified efficiently by fuzzers. Therefore, for testing purposes a lower level language is preferred.

Servers that fulfil the requirements to be open source and written in lower level languages are among others ngIRCd[3] and InspIRCd[5]. For this thesis ngIRCd is chosen as preferred SUT.

[4]https://github.com/ngircd/ngircd/blob/master/doc/Protocol.txt
[5]https://github.com/inspircd/inspircd

NgIRCd is an open source (GPL licensed) IRC server developed in C since 2001. At the time of writing, the latest release is version 26.1. In 2020 some fuzz testing has been done using AFL on version 26 rc2. This fuzz testing has lead to two publicly known vulnerabilities in the S2S communication[6][7], of which one is documented in CVE–2020–14148. These vulnerabilities seem to involve exclusively authenticated S2S communication. Due to the trusted by design nature of S2S communications, these vulnerabilities are currently not prioritised to be resolved by the main software maintainer[8].

Furthermore, seven older CVEs[9] for ngIRCd are documented. These are published between 2005 and 2013 and currently not registered as open issues. Therefore, these CVEs are assumed to be mitigated in the current version of ngIRCd.

[6] https://github.com/ngircd/ngircd/issues/274

[7] https://github.com/ngircd/ngircd/issues/277

[8] https://github.com/ngircd/ngircd/pull/276

[9] Older CVEs for ngIRCd are CVE-2013-5580, CVE-2013-1747, CVE-2009-4652, CVE-2008-0285, CVE-2007-6062, CVE-2005-0199 and CVE-2005-0226.

Chapter 5

Experiments

At the start of this chapter (section 5.1), considerations regarding the overall testing setup are given. With this testing setup, four different fuzzing techniques are compared:

1. A baseline approach based on AFLNet. This approach uses only essential features to enable fuzzing of the IRC protocol and is described in section 5.3.

2. A configuration to AFLNet given in section 5.4 to apply a dictionary based on the IRC specifications as described in chapter 4.

3. Instrumentation of the SUT to provide information about the current state to AFLNet is described in section 5.5.

4. An extension to AFLNet to use checkpointing is described in section 5.6.

An evaluation of the performance of each approach is given based on the evaluation criteria that are discussed in section 5.2. It is expected that the three optimisation approaches (approaches two, three and four) and the combination of these approaches will perform better than the baseline approach. Both, a comparison of the performance of each approach and a combination of the second and third approach are given in section 5.7.

To evaluate the performance of different fuzzing approaches more independent of the SUT, several synthetic bugs are introduced as described in section 5.8.

5.1 Testing setup

The hardware aspects of the testing setup used in this thesis are described in section 5.1.1. Secondly, the expected performance of this setup is evaluated in section 5.1.2.

5.1.1 Testing hardware

Fuzzing is often non-deterministic, which can influence performance. Additionally, performance can be effected by external influences such as other resource intensive programs. To keep benchmarks comparable between different runs of a fuzzing program, a machine with minimal external influences is recommended. As suggested in the documentation of AFLNet, otherwise idle machines running Linux without non-essential programs have been used in this case study.

To be able to repeat experiments sufficiently some experiments are done locally and others on virtual servers. The local experiments are run on several different machines. To reduce load times, all binary and configuration files are stored on a SSD.

The fuzzing process and SUT run locally on multi-core systems. This could allow speeding up the fuzzing process by using several cores in parallel, however this option has not been used. Online multiple virtual machines of Linode are rented[1]. At Linode the rented servers have a single CPU core and one gigabyte RAM available. Faster options with a dedicated CPU are also available. Such servers seem to perform about twice as fast compared to the shared single core servers for fuzzing with AFLNet. However, the price increased six times when upgrading to the more expensive model. For an optimal price-performance ratio, the case study used the cheaper single-core servers.

5.1.2 Expected testing performance

Additional configuration instructions regarding fuzzing performance are provided in the respective AFLNet documentation[2]. Applying the OS configuration recommendations results locally in a peak speed increase of about sixty executions per second from 32 up to 93 executions per second with ngIRCd as SUT. On longer test runs, the execution speed slows down considerably to even below one execution per second.

According to the documentation of AFLNet, around 500 executions per second is preferred[3]. However, this guideline seems to be equal to the

[1] https://www.linode.com

[2] https://github.com/aflnet/aflnet/blob/master/docs/perf_tips.txt

[3] Stage progress in https://github.com/aflnet/aflnet/blob/master/docs/status_screen.txt

guideline of AFL. Due to the overhead of among others (local) network communication, slower executions speeds are expected. Other benchmarks for stateful programs also only reach speeds up to five executions per second [28].

To asses the impact of using a different system than the authors of AFLNet, the execution speed of the example SUT is compared. The Live555 media streaming server[4] is used as an example in the instructions of AFLNet and as SUT in the paper introducing AFLNet [11]. Comparable performance is observed with the streaming and the IRC server. Therefore, it is assumed that differences between the used systems account for execution speed differences. Possible opportunities to analyse and partly mitigate this performance gap in more detail are summarised in section 6.2.

5.2 Evaluation of experiments

To evaluate the results of the experiments, several measures are used. These measures are elaborated in section 5.2.1. To compare the results of several experiments, statistical tests are used. Details regarding the choice of statistical tests are provided in section 5.2.2.

5.2.1 Evaluation Measures

To evaluate the progress and effectivity of the fuzzing approaches, three key elements are required. First, a baseline is used to compare the impact of the tested approaches. This baseline is described in section 5.3.

Second, a metric is needed to compare the tested fuzzer with the baseline. In the literature, several metrics are used as described in section 2.2. Since the experimental setup is based around AFLNet, as performance metrics total execution paths found, maximum sequence depth reached and unique crashes triggered are used.

Fuzzing is a time intensive process. To get meaningful results, it is not unusual that fuzzing still yields new findings after a month of fuzz testing[5]. For this experimental setup, fuzzing is conducted for 500.000 executions each. This roughly corresponds to a four days fuzzing experiment. Recommendations to select an appropriate stopping point based on achieved results are given in section 6.5.

5.2.2 Comparison of experiments

To determine if potential differences between the fuzzing approaches are statistically significant, the effect size of the experiments is calculated based

[4] http://www.live555.com/mediaServer/

[5] https://blog.trailofbits.com/2021/03/23/a-year-in-the-life-of-a-compiler-fuzzing-campaign/

on the results of the base line in section 5.3.4. Additionally, statistical tests are used to compare the results of different approaches.

The results of each approach are compared with the baseline. This can be regarded as two groups of observations. Since, these observations cannot be paired uniquely with observations from another approach, the observations represent independent events.

Each experiment to evaluate an approach is repeated five times to reduce non-structural differences caused by uncontrolled variables in the testing setup or the non-deterministic behaviour of the fuzzer. For each repetition the fuzzer uses the exact same settings, input seed and SUT.

Preferably the same statistical test is used to compare all conducted experiments with the baseline. However, it is not predictable upfront whether all results are normally distributed and of similar variance (homoskedastic).

The Shapiro-Wilk's test is used to check normal distribution. A result of $p \geq 0.05$ implies it is unlikely that the data is not normally distributed. Under the assumption that the data is normally distributed, it is possible to use Welch's t-test instead of the Mann Whitney U-test.

5.3 Baseline with AFLNet

All evaluated approaches are compared with a baseline. As a baseline, AFLNet (see section 3.1) is used. Since AFLNet by default does not support the IRC protocol, a protocol extension is implemented as described in section 5.3.1. For creating the baseline, no further changes are made to the fuzzer. The goal of the baseline experiments is to be able to compare the performance impact of optimalisation approaches. The exact setup of AFLNet is discussed in section 5.3.2. Unless mentioned otherwise, the same configuration parameters are also used in experiments to test additional approaches. Results of the baseline experiment are given in section 5.3.4.

5.3.1 Implementation of the IRC protocol

To fuzz the SUT, AFLNet must be extended to support the IRC protocol. In order to support a new protocol in AFLNet a function has to be implemented to extract responses of the SUT. Additionally, the protocol should be added to the list of supported protocols to be able to actually use a newly implemented protocol.

For the baseline experiment only the function:
`region_t*extract_requests_irc()` has been implemented. Further support for state-aware fuzzing is implemented for the experiments described in section 5.5.

The "request extraction" function has similarities with the function for other already implemented protocols. Therefore, it is possible to base the IRC support on the implementation of the SMTP protocol. The used implementation is provided in listing 1.

5.3.2 Configuration of AFLNet

AFLNet has multiple configuration options. Both runtime parameters and compile time environment variables impact the fuzzing performance of AFLNet. The script used to configure some environment variables and performance settings is provided in appendix A.1. Compile-time settings regarding AFLNet include enabling LLVM and options to catch memory faults.

Run time parameters of AFLNet can be used to control the fuzzing. Two parameters, netinfo and protocol are mandatory. In this experiment netinfo (-N) points to localhost with the default IRC port 6667. The protocol (-P) parameter is set to IRC to use the implemented function described in section 5.3.1.

The following runtime parameters are set to gain reasonable performance:

- Skip deterministic fuzzing steps (-d) is enabled to improve fuzzing performance. This gives quicker results earlier in the fuzzing process,

which allows shorter test runs. The disadvantage of this options is that potentially found crashes are harder to reproduce.

- False negative reduction mode (-F) is activated. According to the documentation of AFLNet, this mode aids AFLNet to distinguish between longer executions and crashes.
- Region-level mutation operators (-R) allow larger mutations to occur. In stateless programs this often improves the fuzzing performance [21]. Therefore this parameter is activated.

Additionally, some runtime parameters are set to improve reproducibility between test runs. These parameters are unchanged in all experiments:

- A server cleanup script (-C) is unspecified. The SUT seems to store no files influencing the state on a subsequent run. Therefore, this parameter is not expected to improve reproducibility.
- Waiting time (-D) specifies in micro seconds how long AFLNet should wait till the SUT is started. For all experiments a timeout of 10000 microseconds is used to improve reproducibility.
- Sending a SIGTERM signal (-K) allows the SUT to shutdown properly. Based on a testing setup, without this parameter AFLNet fails the dry run of the initial seed. This prevents AFLNet from starting to fuzz.
- Both state selection (-q) and seed selection (-s) algorithms are set to "favor".

5.3.3 Configuration of ngIRCd

To apply AFLNet on ngIRCd some minor modification have been made to the ngIRCd binary. The makefiles of ngIRCd are generated using a configuration script. NgIRCd is mainly written in C. Therefore to instrument the SUT, the compiler afl-clang-fast can be used. This compiler is available along with AFLNet and implements coverage feedback-enabled instrumentation. Additionally, LLVM[6] is installed to enable AFLNets LLVM mode[7]. This allows AFLNet to use compiler-level instead of assembly-level instrumentation.

Other configuration changes reduce logging and disable zlib compression for a potential speed-up. The complete configuration command becomes:

```
./configure --without-syslog --witouth-zlib CC=afl-clang-fast   CFLAGS=m32
```

[6] https://llvm.org/

[7] https://github.com/aflnet/aflnet/tree/master/llvm_mode

For this experiment the AddressSanitizer (ASan) is not used. However, if one chooses to use ASan, one should ensure that ngIRCd is built as 32-bit binary. Compiling the binary in 32-bit mode ensures memory usage of ASan remains reasonable. According to the documentation of AFLNet using a 64-bit binary can require up to 20 TB of memory allocations instead of up to 800 MB. The reserved memory can be checked with:

```
ulimit -Sv \$[999 << 10]; ./ngircd-64-asan
```

Using this command shows that the process indeed tries to reserve 15TB when compiled in 64-bit mode. Some background information regarding ASan is provided in section 3.1.

5.3.4 Baseline results

The baseline is evaluated on maximum depth reached, total paths found and unique crashes found as described in section 5.1. This experiment is repeated five times. Each repetition uses the same settings, initial seed and SUT and is terminated after approximately 500.000 executions. Results of each test repetition can be found in table 5.1. The number of paths found and the maximum depth reached per execution are shown in fig. 5.1a and fig. 5.1b. No iteration of this experiment found any bugs.

Intuitively, a correlation is expected between the paths found and the maximum depth reaches. The baseline experiments suggest a weak positive correlation (Pearson correlation coefficient 0.19), however this relation is not statistically significant ($p \approx 0.76$). This weak correlation might be caused by the findings in experiment 0 and 2. For these two experiments less paths are found when a higher depth was reached.

Effect size

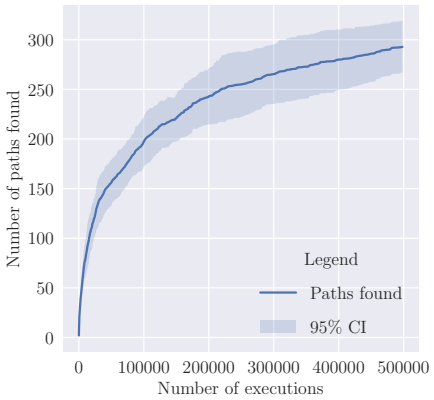
Based on analysis of the baseline result, the measured effect size is calculated using eq. (5.1) [29]. This effect size allows to reason whether a difference between experiments is a real effect or caused by a measurement error. Intuitively, if more experiments are conducted, less variation between experiments occurs. Alternatively, if more errors are accepted, it would be possible to measure smaller differences between the different fuzzing approaches with the same sample size. To adhere to common practice the type-1 error probability (α) is set to 5%. The type-2 error (β) is set to 20% [30][31]. And, the population standard deviation is estimated using the conducted experiments as samples.

$$ES = \sqrt{2 \cdot \frac{\sigma^2 (z_\alpha - z_\beta)^2}{n}} \tag{5.1}$$

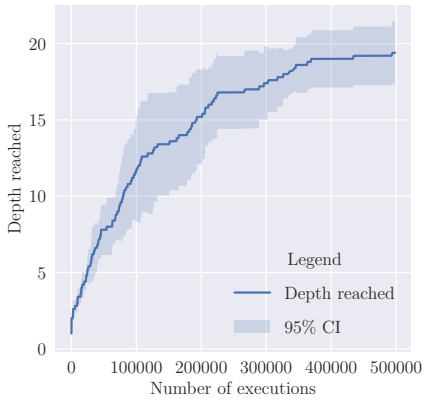
	Max Depth	Total Paths
Exp. 0	20	260
Exp. 1	19	282
Exp. 2	16	296
Exp. 3	21	296
Exp. 4	21	331
Average	19	293
Standard deviation	2	23
Normal distribution p-value	0.17	0.79

Table 5.1: Results of the five baseline test runs after approximately 500.000 executions. Each test case makes use of the same seed and runtime parameters. Additionally the average, standard deviation and the p-value of the Shapiro-Wilk's test is provided. Based on the Shapiro-Wilk's test each measurement of the baseline is normally distributed ($p \geq 0.05$). No test run found any vulnerabilities, therfore these data are not shown.

Based on this calculation, a difference of approximately 114 paths found and a reached maximum depth of 11 in the SUT can be measured.



(a) The average number of paths found with the standard deviation.



(b) The maximum depth reached with the standard deviation.

Figure 5.1: Visualisation of the number of paths found and maximum depth reached with the base line experiments. The detailed results per experiment are provided in table 5.1

```
region_t* extract_requests_irc(unsigned char* buf, unsigned int buf_size,
                               unsigned int* region_count_ref) {
  char *mem;
  unsigned int byte_count = 0;
  unsigned int mem_count = 0;
  unsigned int mem_size = 1024;
  unsigned int region_count = 0;
  region_t *regions = NULL;
  char terminator[2] = {0x0D, 0x0A};
  mem = (char *)ck_alloc(mem_size);
  unsigned int cur_start = 0;
  unsigned int cur_end = 0;
  while (byte_count < buf_size) {
    memcpy(&mem[mem_count], buf + byte_count++, 1);
    //Check if the last two bytes are the IRC message terminating bytes
    if ((mem_count > 1) && (memcmp(&mem[mem_count - 1], terminator, 2) == 0)) {
      region_count++;
      regions = (region_t *)ck_realloc(regions, region_count * sizeof(region_t));
      regions[region_count - 1].start_byte     = cur_start;
      regions[region_count - 1].end_byte       = cur_end;
      regions[region_count - 1].state_sequence = NULL;
      regions[region_count - 1].state_count    = 0;
      mem_count = 0;
      cur_start = cur_end + 1;
      cur_end = cur_start;
    } else {
      mem_count++;
      cur_end++;
      if (cur_end == buf_size - 1) {       //Check if the last byte has been reached
        region_count++;
        regions = (region_t *)ck_realloc(regions, region_count * sizeof(region_t));
        regions[region_count - 1].start_byte     = cur_start;
        regions[region_count - 1].end_byte       = cur_end;
        regions[region_count - 1].state_sequence = NULL;
        regions[region_count - 1].state_count    = 0;
        break;
      }
      if (mem_count == mem_size) { // Double the buffer size
        mem_size = mem_size * 2;
        mem=(char *)ck_realloc(mem, mem_size);
      }
    }
  }
  if (mem) ck_free(mem);
  //in case region_count equals zero, it means that the structure of the buffer is broken
  //hence we create one region for the whole buffer
  if ((region_count == 0) && (buf_size > 0)) {
    regions = (region_t *)ck_realloc(regions, sizeof(region_t));
    regions[0].start_byte = 0;
    regions[0].end_byte = buf_size - 1;
    regions[0].state_sequence = NULL;
    regions[0].state_count = 0;
    region_count = 1;
  }
  *region_count_ref = region_count;
  return regions;
}
```


Listing 1: Implementation of the message parse function in AFLNet for the IRC protocol.

5.4 Grammar based fuzzing for the IRC protocol

A grammar can improve fuzzing of protocol syntax. Specifying syntactically correct input mutations can potentially increase coverage while fuzzing. More background information regarding grammar based fuzzing is provided in section 2.3. Grammar based fuzzing with an upfront specified grammar is not supported in AFLNet. However, an alternative in a simple form is provided with dictionaries. More information regarding the dictionary implementation can be found in section 5.4.1. In section 5.4.2 the results are summarised.

5.4.1 Dictionary implementation

AFLNet can be provided with a dictionary. A dictionary is conceptually a list of keywords that is used to mutate messages. Such a dictionary lacks specifications of protocol structure. Therefore, created messages do not always adhere to protocol constraints. For instance, a channel name should always follow after a new channel command. However, mutations that are rejected due to incorrect syntax, will often quickly be discarded by the fuzzer[8]. This concept allows to approximate a grammar at run time. Another limitation when using a dictionary instead of a more expressive grammar is the missing option to recurse or expand rules.

A dictionary can be specified in two formats. A single keyword can be stored in each file. In this case an entire directory forms a single dictionary. Alternatively, a single file can be used with a list of keywords between quotations marks. Special and unprintable characters have to be represented in a hexadecimal ASCII value. Single digit hexadecimal numbers also need an additional zero as prefix.

For the IRC protocol a set of commands and special characters is extracted from the RFC and ngIRCd[9] specifications. These keywords are used to create a dictionary of the second single file format. This dictionary is shown in Figure 5.2. In this dictionary several characters with a special meaning in the IRC protocol and commands of the IRC protocol are included.

5.4.2 Dictionary experiment results

A higher performance is expected compared to the baseline experiments in section 5.3.4. The results are summarised in table 5.2. Graphical representations of the paths found and maximum depth reached per execution are shown in fig. 5.3a and fig. 5.3b.

[8] https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html
[9] https://github.com/ngircd/ngircd/blob/master/doc/Commands.txt

```
#                    "GLINE"               "NAMES"
# AFL Dictionary for "KLINE"               "LIST"
# the IRC protocol   "SERVICE"             "INVITE"
"\x00"               "SERVLIST"            "KICK"
"\x07"               "SQUERY"              "VERSION"
"\x15"               "SVSNICK"             "STATS"
"\x012"              "CHANINFO"            "LINKS"
" "                  "METADATA"            "TIME"
","                  "GET"                 "CONNECT"
"@"                  "POST"                "TRACE"
"#"                  "NJOIN"               "ADMIN"
"&"                  ":"                   "INFO"
"$"                  "ERR_NONICKNAMEGIVEN" "PRIVMSG"
"HELP"               "ERR_NICKNAMEINUSE"   "NOTICE"
"CAP"                "ERR_ERRONEUSNICKNAME""WHO"
"CAP LS"             "ERR_NICKCOLLISION"   "WHOIS"
"CAP LIST"           "ERR_NEEDMOREPARAMS"  "WHOWAS"
"CAP REQ"            "ERR_ALREADYREGISTRED""KILL"
"CAP ACK"            "USER"                "PING"
"CAP NAK"            "SERVER"              "PONG"
"CAP CLEAR"          "ERR_ALREADYREGISTRED""ERROR"
"CAP END"            "OPER"                "AWAY"
"CHARCONV"           "ERR_NEEDMOREPARAMS"  "REHASH"
"PASS"               "ERR_NOOPERHOST"      "RESTART"
"NICK"               "RPL_YOUREOPER"       "SUMMON"
"WEBIRC"             "ERR_PASSWDMISMATCH"  "USERS"
"LUSER"              "QUIT"                "WALLOPS"
"MOTD"               "SQUIT"               "USERHOST"
"PART"               "JOIN"                "ISON"
"DIE"                "MODE"
"DISCONNECT"         "TOPIC"
```

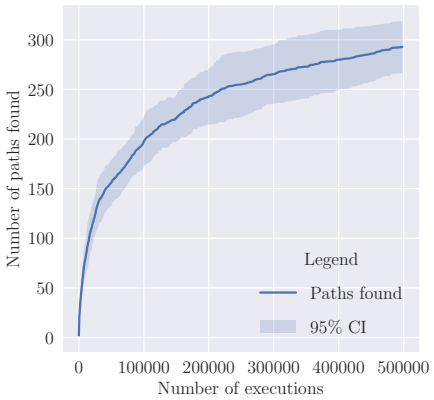
Figure 5.2: Dictionary for the IRC protocol.

Contrary to the baseline experiments, when using a dictionary there appears to be an insignificant ($p \approx 0.49$) negative correlation (pearson $= -0.51$) between max depth and paths found. Partly, this can be explained by experiment 0. This experiment reaches an approximately two standard deviations higher max depth compared to the average. At the same time, this experiment has a slightly below average total paths found. This could indicate that the sample size is too low, further research is required to investigate if there is indeed a negative correlation.

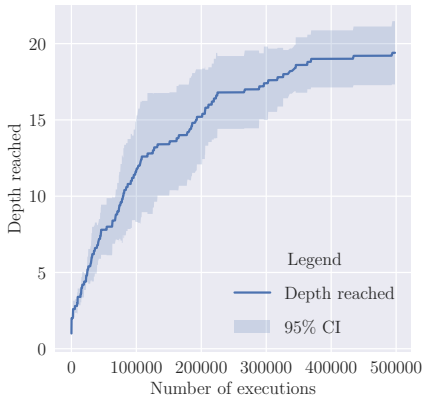
Additionally, the cut-off point of 500.000 executions might be not sufficient to find a correlation. It seems that a fuzzer with a dictionary still frequently finds new paths after around 500.000 executions.

	Max Depth	Total Paths
Exp. 0	44	449
Exp. 1	18	514
Exp. 2	25	417
Exp. 3	21	450
Exp. 4	27	479
Average	27	462
Standard deviation	9	33
Normal distribution p-value	0.21	0.84

Table 5.2: Results of the five test runs using a dictionary after approximately 500.000 executions. Each test case makes use of the same initial seed and runtime parameters. Additionally the average, standard deviation and results of the Shapiro-Wilk's test is provided. Based on the Shapiro-Wilk's test each measurement of the baseline is normally distributed ($p \geq 0.05$). No test run found any vulnerabilities, therfore these data are not shown.



(a) The number of paths found with the standard deviation.



(b) The maximum depth reached with the standard deviation.

Figure 5.3: Visualisation of the number of paths found and maximum depth reached with the dictionary experiments. The results per experiment are provided in table 5.2.

5.5 State-aware fuzzing

A SUT can be divided into different states. The state of a SUT depends on the content of the program's memory and is influenced by sending instructions to the SUT. By extracting information regarding the current state it is possible to target a specific states of the SUT. During fuzzing, AFLNet can use this information to learn the state machine of the SUT.

To aid implementing state-aware fuzzing and to be able to verify the results, a state machine has been constructed for the IRC protocol. This state machine can be found in section 5.5.1.

Details regarding the implementation of state-aware fuzzing for the IRC protocol are provided in section 5.5.2. Results of the experiment can be found in section 5.5.3.

5.5.1 State machine of the IRC protocol

A state machine is not provided in the IRC specifications. Therefore, a manual construction of the state machine is conducted based on the IRC protocol specified in RFC 2812 [25]. Graphviz [32] has been used to generate the state machine.

This machine is visualised as graph and shown in Figure 5.4. In this graph a node is defined as a state of the IRC protocol and an edge as transition between states. Each edge represents one or more requests. Any request can potentially fail, which results in an error response and no change of state. These connections were removed from the visualisation. Also, the authenticated commands Mode, Topic, Names, List, Invite, Kick, Privmsg, Notice, Motd, Luser, Version, Stats, Links, Time, Connect, Trace, Admin, info, Serverlist, Squery, Who, Whois, Whowas, Kill, Ping, Pong, Away, Rehash, Die, Restart, Summon, Users, Oper, Operwall, Userhost and Ison are combined in the edges labelled as "Queries".

In this state machine three high level phases are identified.

- The authentication phase where the client connects to the server. In this phase the SUT responses with the CAP command. CAP commands are used to negotiate server functionality. This phase is normally ended by the client by sending the CAP END command.
- The messaging phase consists of most IRC commands. During this phase for instance channels could be joined or permissions escalated.
- The termination phase should start after the SUT receives a (S)Quit message. The only expected response would be a NOTICE command that the client is disconnected.

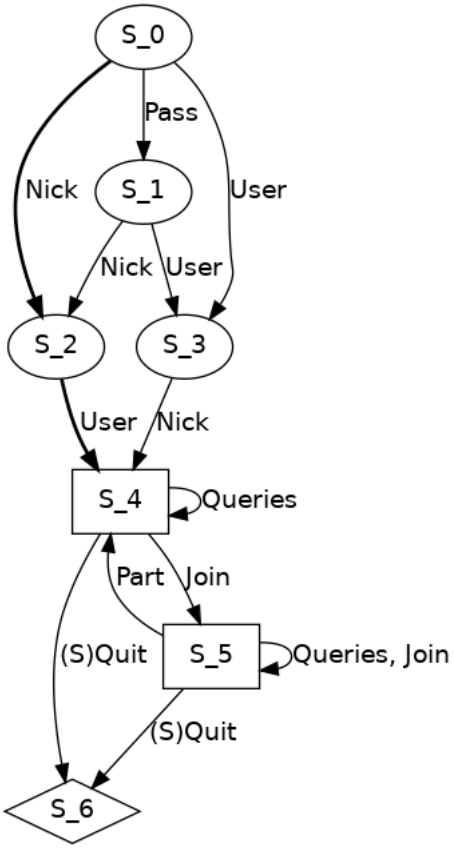


Figure 5.4: Manually constructed state machine of IRC protocol based on RFC 2812. There are tree phases identified in the protocol. The authentication phase (ellipses), the messaging phase (boxes) and the termination phase (diamond). The expected flow in the authentication phase has been annotated by bold arrows.

5.5.2 Implementation of state-aware fuzzing

To implement state-aware fuzzing, the fuzzer has to determine the current state of the SUT. Some information regarding the state of the SUT can be inferred based on received responses. Additionally, by instrumenting the SUT, it is possible to receive a response code containing a more precise indication of the current state of the SUT. To identify different states in the SUT, the state machine constructed in section 5.5.1 can be used to analyse the protocol.

Based on these states the SUT could be instrumented. In total 381 different places in the SUT use a function declared in `irc-write.h` to send send a IRC message. Some of these messages are only send to other servers. The testing setup uses only a single SUT. Therefore this part of the func-

tionality would not need to be instrumented. The only changed function calls are regarding `IRC_WriteStrClient` and `IRC_WriteErrClient`. With this information, the fuzzer could determine its target state more precisely. However, this approach could not be used on a different IRC server as SUT, since this would require each SUT to be instrumented manually.

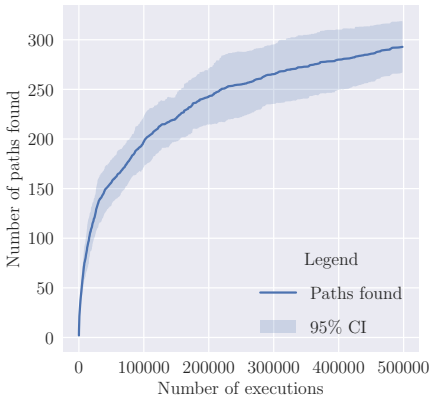
Alternatively, without instrumentation of the SUT, states can be estimated by AFLNet by implementing `extract_response_codes_irc`. IRC instructions can only be validly used in one of the three high level phases each defined in section 5.5.1. Using a numerical value of each command as state information can therefore be used as an approximation that is independent of the SUT. Therefore this approach could be used to test any IRC server without additional manual work. This method is used to implement state-aware fuzzing for the case study. The implementation can be found in listing 2.

5.5.3 State-aware experiment results

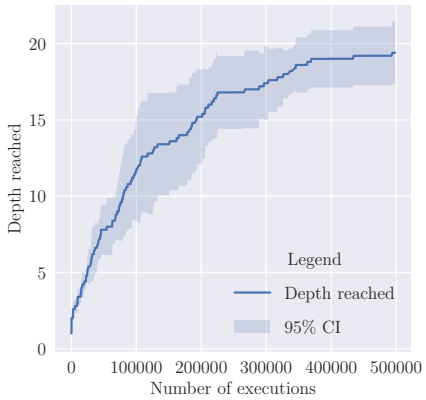
The results of each test repetition with state-aware fuzzing can be found in table 5.3. The number of paths found and maximum depth reached per execution are shown in fig. 5.5a and fig. 5.5b. Similary to the baseline, there is a positive correlation between the depth reached and the paths found. This correlation is also statistically insignificant ($p \approx 0.14$), but has a high correlation coefficient of 0.76. With a larger sample size, a statistically significant correlation is likely to occur.

	Max Depth	Total Paths
Exp. 0	21	342
Exp. 1	13	305
Exp. 2	15	283
Exp. 3	13	321
Exp. 4	22	347
Average	17	320
Standard deviation	4	24
Normal distribution p-value	0.11	0.69

Table 5.3: Results of the five state-aware test runs after approximately 500000 executions. Each test case makes use of the same seed, parameters and SUT. Additionally the average, standard deviation and results of the Levene test is provided. Based on the Levene test each measurement of the baseline is normally distributed ($p \geq 0.05$). No test run found any vulnerabilities, therfore these data are not shown.



(a) The number of paths found with the standard deviation.



(b) The maximum depth reached with the standard deviation.

Figure 5.5: Visualisation of the number of paths found and maximum depth reached with the state-aware experiments. The results per experiment are provided in table 5.3.

```
unsigned int* extract_response_codes_irc(unsigned char* buf, unsigned int buf_size,
                                         unsigned int* state_count_ref)
{
  char *mem;
  unsigned int byte_count = 0;
  unsigned int mem_count = 0;
  unsigned int mem_size = 512;
  unsigned int *state_sequence = NULL;
  unsigned int state_count = 1; //  Allocate room for the sequence
  const char terminator[2] = {0x0D, 0x0A};
  const char space[1] = {0x20};
  mem=(char *)ck_alloc(mem_size);
  state_sequence = (unsigned int *)ck_realloc(state_sequence,
                                              state_count * sizeof(unsigned int));
  state_sequence[state_count - 1] = 0;

  while (byte_count < buf_size) {
    memcpy(&mem[mem_count], buf + byte_count++, sizeof(char));

    if ((mem_count > 0) && (memcmp(&mem[mem_count - 1], terminator, 2) == 0)) {
      char temp[5];
      memcpy(temp, mem, 2);
      int incr = 0;
      if(temp[0] == 0x3a){
        while ((incr < mem_count) && (memcmp(&mem[incr], space, 1) != 0)) {
          incr++;
        }
      }
      memcpy(&temp[0], &mem[incr], 5);
      unsigned int message_code = (unsigned int) atoi(temp);
      state_count++;
      state_sequence = (unsigned int *)ck_realloc(state_sequence,
                                                  state_count * sizeof(unsigned int));
      state_sequence[state_count - 1] = message_code;
      mem_count = 0;
    } else {
      mem_count++;
    }
  }
  if (mem){ ck_free(mem); }
  *state_count_ref = state_count;

  return state_sequence;
}
```

Listing 2: Implementation of the response extraction function in AFLNet for the IRC protocol.

5.6 Checkpointing the SUT

Stateful fuzzing uses a sequence of input messages to test the SUT. Depending on the target, long sequences are required to fuzz a deep state. Checkpointing aims to increase fuzzing speed by restoring a message at the end of a sequence instead of repeating the same sequence. More background information regarding the concept of checkpointing in the context of stateful fuzzing is provided in section 5.6.1. To implement checkpointing, several approaches were evaluated. The first two approaches mentioned in section 5.6.2 attempted to use containerisation and were based on Docker and Podman. Due to technical constraints, this approach cannot be used without extensive changes to AFLNet. An alternative approach using DMTCP is introduced in section 5.6.3. This approach does not have the same limitations as the containerisation approach.

5.6.1 Checkpointing a stateful protocol

Checkpoints enable saving the complete state of the SUT after several executions. This prevents repeating longer message sequences and therefore reduces duplication while fuzzing. An example sequence with four messages is shown in Figure 5.6. In this example s_4 is the target state. To reach this state, usually the sequence m_1, m_2, m_3 has to be repeated before m_4 can be used to fuzz state s_4. With checkpointing it would be possible to save s_3 after visiting this state. To fuzz s_4, a checkpoint of s_3 could ideally be restored to prevent sending the entire message sequence repeatedly.

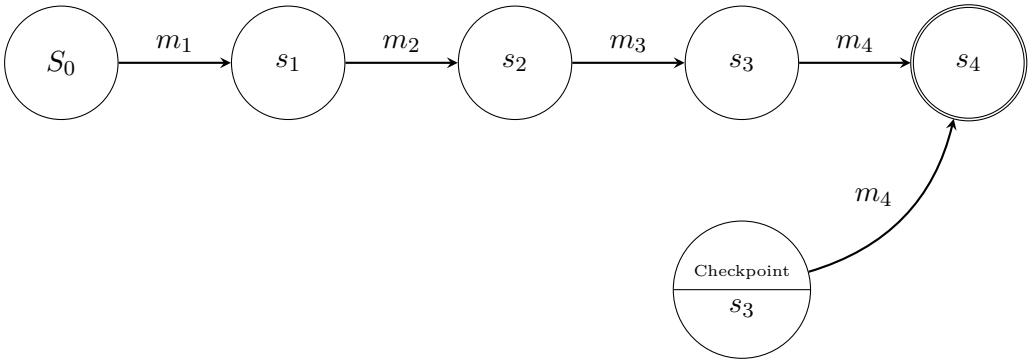


Figure 5.6: Visualisation of the optimal benefit of checkpointing with a four message long sequence. In this graph S_0 is the starting state and S_4 the final state.

5.6.2 Checkpointing with containers

Containers can be checkpointed to restore a previous state of the SUT. Such checkpoints can be used without modifications to the SUT. The SUT has to run in a "container", which is an isolated user space that shares the kernel with the host operating system[10]. Crashes of the SUT are monitored using core dumps. A core dump is managed by the kernel[11], therefore crash monitoring is expected to work even when the SUT is containerised. Some background information regarding the containerisation platform choice is provided in this section. An attempt to implement checkpointing with the chosen container platform Podman is described below.

Containerisation platform

A popular platform to containerise applications is Docker[12]. However, the current Docker API does not provide checkpoint functionality[13]. An alternative containerisation platform, Podman, does provide this functionality[14]. Therefore, Podman is used for the experiments regarding checkpointing by containerisation.

Due to an inconsistency in the restore API of Podman, exporting a checkpoint to an external archive using the API seems currently not possible[15]. This functionality could be used to fuzz multiple versions of the SUT in parallel. Without this functionality it is still possible to create a checkpoint of a SUT and restore this serially at a later moment.

Besides checkpointing, Podman offers a similar functionally called exports. For this use case exporting is not usable. The key difference between checkpoints and exports is that the entire state of all processes is saved for a checkpoint. For an export, only the file system is saved. Therefore a restored export will not be completely equivalent to repetition of a message sequence.

To estimate the feasibility of this approach to checkpointing, the restore time of a container using the Podman API is measured. A full checkpoint-restore cycle requires $2.2\,\mathrm{s} \pm 0.1$. More details regarding the measurement setup are provided in appendix A.2.

Podman approach

To integrate checkpointing with AFLNet, modifications to the fuzzer are required. Primarily, checkpoints have to be created after finding new states.

[10] https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters

[11] https://man7.org/linux/man-pages/man5/core.5.html

[12] https://www.docker.com/

[13] https://docs.docker.com/engine/api/v1.41/

[14] https://docs.podman.io/en/latest/_static/api.html

[15] https://github.com/containers/podman/issues/6517

Additionally, these checkpoints should be used to restore a state instead of replaying a message sequence. AFLNet uses two approaches to choose a target, either with or without state-aware mode. Therefore, implementing a restore functionality should ideally occur for both the state-aware and regular mode. In this thesis, state-aware targeting a state has been extended with the implementation of checkpointing. This could allow testing of a combined improvement of checkpointing and state-aware fuzzing as implemented in section 5.5.

To interact with the Podman API libcurl has been used[16]. Libcurl provides a C API to make synchronous requests. Synchronous requests will halt the program until in this case a checkpoint is either created or restored. For testing purposes a function is added to prevent duplicating code. This function is shown in listing 3.

By default AFLNet uses a fork server to quickly start a new version of the SUT. To implement a minimal working concept of containerised checkpointing, requests have been added to `run_target()`. An implementation using this approach can successfully start, checkpoint and restore a SUT. For testing purposes a container for the SUT is build using the Dockerfile provided in listing 4.

However, this method turned out not to work with the instrumentation of AFLNet. Instrumentation could not be observed by the fuzzer anymore. This issue prevented, in the contexts of this thesis, the usage of AFLNet as greybox fuzzer with containerised checkpoints. A possible solution could be interpreting instrumentation in each container independently and sending this data back to the host process.

5.6.3 Checkpointing with DMTCP

Another method to checkpoint a program is with Distributed MultiThreaded Checkpointing (DMTCP) [33]. This program can checkpoint programs on Linux systems written in many frequently used languages such as C, C++ and Python.

Research has been conducted previously on using DMTCP to optimise generating state machines of programs [34]. This task, similary to fuzzing, benefits from a high coverage of the SUT. Measurements indicate that DMTCP can positively impact performance while exploring a stateful program.

Implementing checkpointing with DMTCP instead of Podman checkpoints would allow AFLNet to observe instrumentation of the SUT. On a high level, DMTCP checkpointing will require adjustments when starting the SUT, storing a SUT state and seeking to a target state.

Implementing DMTCP is preferred to be based on the execve mode

[16] https://curl.se/libcurl/

```
u32 post_request(const char url[100], const char params[100]){
  u32 returnv = 0;
  CURL *curl;
  CURLcode res;
  curl_global_init(CURL_GLOBAL_ALL);
  curl = curl_easy_init();

  if(curl){
    curl_easy_setopt(curl, CURLOPT_URL, url);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, params);
    res = curl_easy_perform(curl);
    if(res != CURLE_OK){
      fprintf(stderr, "curl post_request() failes: %s\n",
              curl_easy_strerror(res));
      returnv=1;
    }
    curl_easy_cleanup(curl);
  } else{
      returnv=1;
  }
  curl_global_cleanup();

  return returnv;
```

Listing 3: To test the integration with the Podman API, a function to write synchronous post requests has been used based on the examples provided in the libcurl documentation.

```
FROM fedora
COPY ./ngircd ngircd
RUN chmod +x ngircd
COPY ./ngircd.conf /usr/local/etc/ngircd.conf
EXPOSE 6667
CMD [ "./ngircd", "-np" ]
```

Listing 4: The dockerfile used to build a container containing ngIRCd. For this container, ngIRCd has been compiled previously with instrumentation. The default generated configuration file has been used.

of AFLNet. By default AFLNet uses a fork server to improve execution speed[17]. More research is needed to evaluate the possible advantages of using principles of the AFLNet fork server in combination with checkpoints. Another consideration when integrating DMTCP in AFLNet is that DMTCP appears to depend on a shell environment. Therefore, while calling DMTCP commands using `system()` is functional, using `execv()` is malfunction.

When saving a state, the state id could be used as a reference to the checkpoint. It should be noted that the state id used by AFLNet is not guaranteed to be unique. However, the likelihood of a collision is expected to be low enough to not negatively impact fuzzing performance in general. A checkpoint could be loaded if a state is targeted, that has been reached before. A possibility to prevent a continuous growth of checkpoints, is to prune checkpoints of states that can be reached with only a fraction of the maximum depth of messages. Further research is needed to find a guideline when a checkpoint should be pruned. This depends on the target state selection algorithm, speed of the SUT and overhead of the checkpoint.

[17] https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

5.7 Comparison of approaches

In this section results of the four previously introduced approaches are compared with the baseline described in section 5.3. An overview of paths found and depth reached in all four approaches can be seen in fig. 5.7. A in-depth comparison of dictionary-based fuzzing is provided in section 5.7.1. More details regarding state-aware fuzzing are given in section 5.7.2. The combination of both approaches is discussed in section 5.7.3. Since for checkpointing the SUT as described in section 5.6, no working implementation is available, this approach has not been included in this section. A summary of all results can be found in table 5.4.

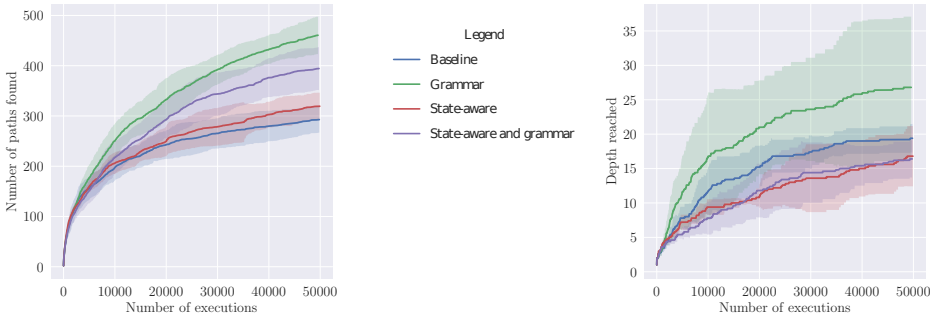
Approach	Max Depth (avg±std)	Total paths(avg±std)
Baseline	19 ± 2	293 ± 23
Dictionary	27 ± 9	462 ± 33
State-aware	17 ± 4	320 ± 24
State-aware with dictionary	16 ± 3	395 ± 38

Table 5.4: High level overview of results from all four experimental approaches. The average and standard derivation of the maximum depth and total paths is provided.

5.7.1 Dictionary based fuzzing

From this overview, dictionary based fuzzing finds on average the most paths. Compared to the baseline almost 60% more paths are found. Intuitively, the confidence interval of the grammar and baseline do not overlap, indicating a considerable improvement. Statistically, this difference is significant ($p \approx 0.00003$) when applying Welch's t-test. The maximum reached depth is on average 42% non-significantly increased ($p \approx 0.14$). As shown in fig. 5.7b, the standard deviation of the depth reached is large, and the 95% confidence interval overlaps with the standard deviation of the baseline. This implies that on average longer message sequences are tested and an experiment with more power could show this statistically.

Based on these results a dictionary appears to be essential for fuzzing of the IRC protocols. This improvement appears to be a result of more syntactically distinguishable mutations, because each mutation used keywords and tokens of the IRC protocol. This principal can be applied to any structured input, therefore, this improvement is likely to occur in any protocol with an extensive syntax. More details regarding fuzzing with a dictionary can be found in section 2.3.



(a) An overview of the number of paths found per execution.

(b) An overview of the depth reached per execution.

Figure 5.7: An overview of the results of the experiments performed to assess the performance of the baseline, grammar and state-aware approach. Additionally, a combination of the grammar and state-aware mode has been evaluated.

5.7.2 State-aware fuzzing

Also in fig. 5.7a, state-aware fuzzing appears to be an improvement on path coverage to the baseline. This improvement is however less then 10% compared to the baseline. On the other hand, state-aware fuzzing decreases the depth reached, on average the maximum depth decreases with approximately 10%. Neither difference is statistically significant. More background information regarding state-aware fuzzing is provided in section 5.5.

An explanation for this negative result could be that the IRC protocol does not have many different states. As shown in section 5.5.1 the protocol can be divided in roughly three main states. Both, the initial authentication state and the final termination state are relatively small. Most IRC commands are in the authenticated state. In this state commands can be used in almost any order, which might reduce the effect of state-aware fuzzing.

Another observation is that the by AFLNet learning state machine is different compared to the state machine designed based on the IRC specifications. Based on the designed state machine, multiple different states can be reaches from most states. However, the learned state machine only branches to several different states from the initial state and continuous without branching. This can be seen in fig. 5.8a. The depth of this state diagram does not correspond with the length of the initial message sequence. This could imply that the fuzzing duration used in this experiment was too short. With more executions, likely more state transitions would be uncovered.

Figure 5.8b shows a visualisation of a state machine generated when combining state-aware fuzzing with a dictionary. In this combination more

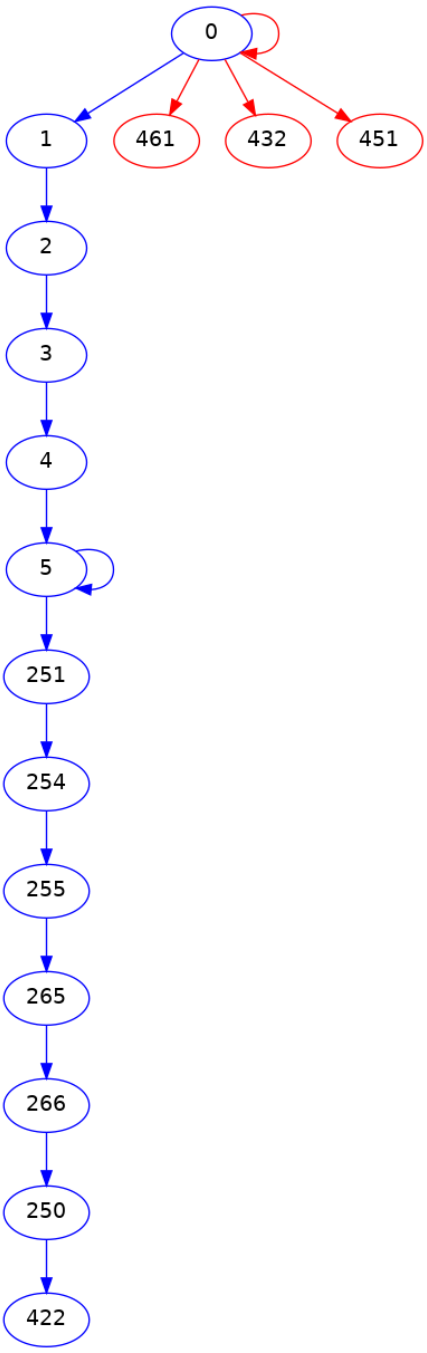
transitions from the initial state are found.

As shown in fig. 5.9, known numeric replies are replaced with full response names. When comparing this state machine with the previously constructed state machine in section 5.5.1, most transitions are expected. The generated diagrams consist of server responses for the queries described mainly in the authentication phase of the protocol. State “265” and “266” are not identified and RPL_STATSDLINE is not part of the RFC analysed in chapter 4. “ERR_NOMOTD” is a response of the Motd query available in the messaging phase. The red coloured states are error responses that are by design omitted in the constructed state machine.

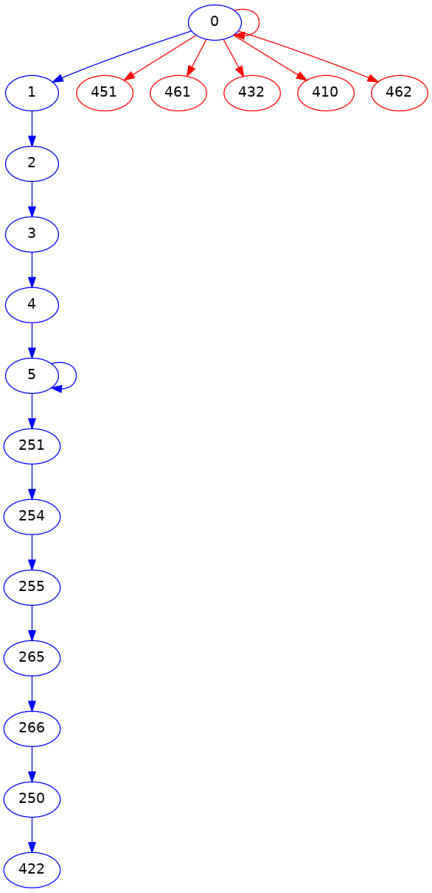
Overall it appears that the messaging and termination phase are not fuzzed optimally. This strengthened the notion that the fuzzing duration of the experiments is too short.

5.7.3 Combining dictionary based and state-aware fuzzing

Finally, a combination of state-aware and dictionary based fuzzing is tested. Surprisingly, this method performed worse compared to only using a dictionary. A possible explanation for this effect could be that parts of a dictionary are only relevant in certain states, which could result in more semantically wrong mutations by using syntax only applicable in different states. Further research is required to find out whether this effect holds true after more iterations.



(a) Generated in state-aware mode.



(b) Generated in state-aware mode with a dictionary provided.

Figure 5.8: State machines generated by AFLNet.

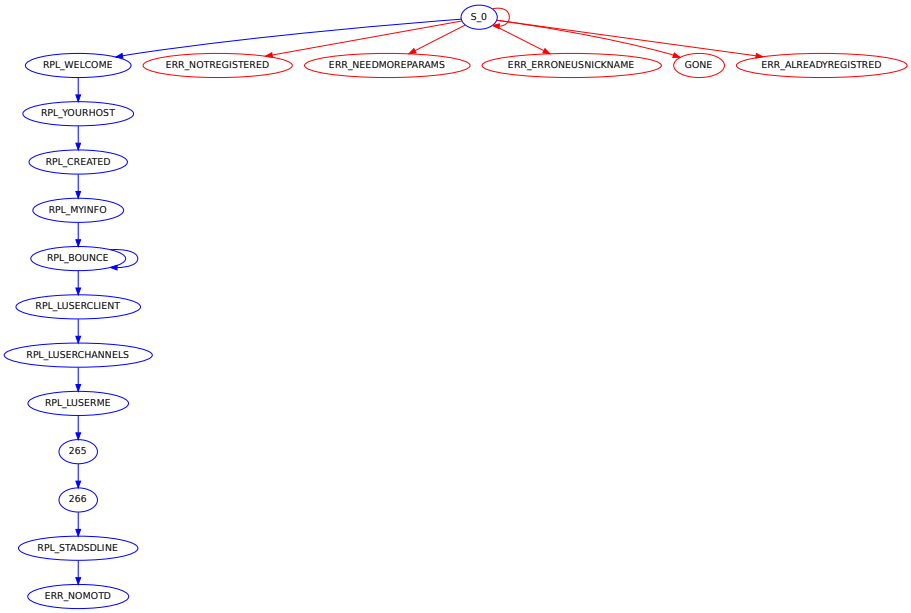


Figure 5.9: Generated in state-aware mode with a dictionary provided using AFLNet.

5.8 Bug Injection

Fuzzing can be used to show that vulnerabilities are present in a SUT. However, if no vulnerabilities are found this does not necessarily imply that the SUT contains no vulnerabilities. To assess if bugs are not found due to a structural problem or that no detectable bugs are present, manual bug injection has been applied.

For this experiment, eight bugs are introduced in several locations of the SUT. Several domain specific vulnerabilities are introduced. For instance, by accepting invalid channel names or allowing users to be kicked multiple times out of a channel. Other bugs consists of adjusting boundary conditions and buffer overflows. A full list of changed functions is provided in table 5.5.

File	Line number	Function name	Change summary
channel.c	656	Channel_IsValidName	invalid channel names are accepted
channel.c	434	Channel_Quit	buffer overflow in enhanced privacy mode
channel.c	354	Channel_Kick	user can be kicked out of a channel independent of if the user is in the channel
channel.c	270	Channel_Join	user can be added to the same channel multiple times
channel.c	148	Channel_InitPredefined	boundary condition changed
conn-func.c	130	Conn_ClearFlags	boundary condition changed
conn-func.c	169	Conn_Next	boundary condition changed
irc-write.c	503	Send_marked_connections	buffer size reduced

Table 5.5: An overview of the eight manually injected bugs in the SUT.

Due to time considerations, testing with artificially introduced bugs is only conducted with 300.000 iterations. Based on the comparison of approaches in section 5.7, dictionary based fuzzing appears to be the most promising approach. Therefore, bug injection is tested with the dictionary based fuzzer described in section 5.4. Each experiment has been repeated five times. Each repetition uses the same settings, seed and SUT. Surprisingly, only in some experiments crashes resulting by the introduced bugs are found.

A summary of the results of these experiments are shown in table 5.6. The average depth is similar to previous experiments. The number of paths found is on average 110% higher compared to the dictionary based approach results in section 5.4.2. This increase is likely caused by injecting bugs.

These experiments allow to measure correlation between the different metrics. There is no significant correlation between the maximum depth reached and the number of bugs found. The Pearson correlation coefficient is 0.04 with $p = 0.94$. There is a stronger positive correlation (0.7) with $p = 0.18$ between the total paths found and number of bugs found. This correlation is still statistically insignificant. However, a large sample size is expected to show a significant relation. Graphically, the correlation between all three metrics is shown in fig. 5.10.

	Max Depth	Total paths	Unique crashes
Exp. 0	16	1011	20
Exp. 1	21	901	0
Exp. 2	26	1086	13
Exp. 3	19	915	2
Exp. 4	16	986	1
Average	20	980	7
Standard deviation	4	67	8
Normal distribution p-value	0.36	0.67	0.15

Table 5.6: Results of the five test runs with artificial bugs after approximately 300.000 executions. Each test case makes use of the same seed and runtime parameters. Additionally the average, standard deviation and the p-value of the Shapiro-Wilk's test is provided. Based on the Shapiro-Wilk's test each measurement of the baseline is normally distributed ($p \geq 0.05$).

Based on the correlation coefficients found in this experiment, total paths found is a stronger indicator of the performance of the fuzzer than the maximum depth reached. Additionally, the fuzzing setup is able to find vulnerabilities, which validates the fuzzing approach used in the previous case studies.

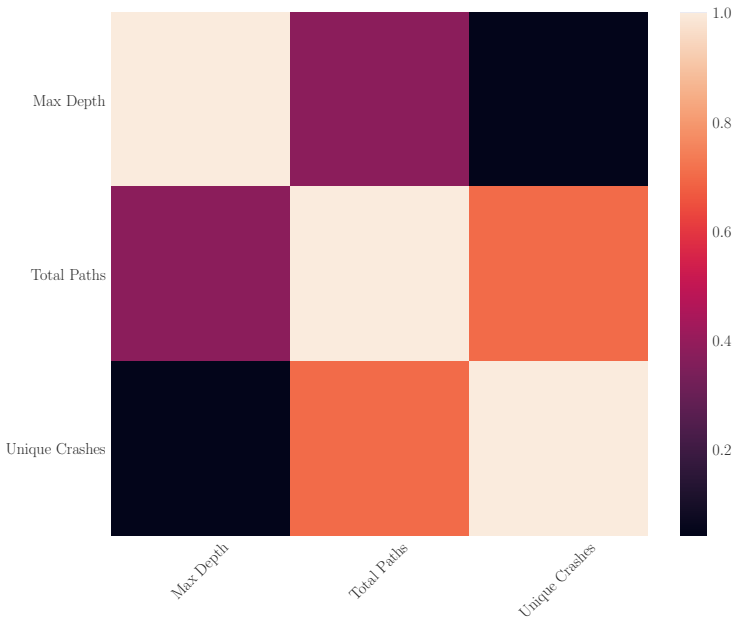


Figure 5.10: The correlation between the maximum depth reached, total path found and unique crashes are shown. In this diagram only the correlation coefficient is shown, $p > 0.05$ for each correlation. The strength correlation is colour coded between weak correlations (0) and strong correlations (1).

Chapter 6

Future work

In the following sections, recommendations for further research to optimise fuzzing are summarised.

6.1 Correlating metrics

In general, one could assume that finding more paths in a SUT and fuzzing longer sequences helps to find bugs. Total paths found is a metric sometimes used in the literature [10] to asses performance.

In chapter 5 three metrics (total paths found, maximum depth reached and number of bugs found) were used to evaluate the performance of the fuzzing approach. However, the power of the conducted experiments was too low to find significant correlations between the used metrics. Therefore, it is suggested to investigate the correlation between number of bugs found and other metrics in more detail. Another possible metric would be measuring code coverage depending on the age of the code. More recent and therefore potentially less mature code could be prioritised.

6.2 Improving execution speed

As described in section 5.1, fuzzing seems relatively slow compared to the execution speed mentioned in the original documentation and in scientific literature. In addition to the the methods used in this thesis, further suggestions to improve execution speed are provided in the AFLNet documentation. For instance, improvements might be gained by profiling and optimising the SUT. Also, brute force protections should be identified and circumvented before fuzzing.

Disk access is also another potential bottleneck regarding execution speed. To partly mitigate this it might be possible to replace safe memory methods with faster alternatives. In this case both the impact on performance and stability of the SUT have to be considered.

6.3 Supporting more complex protocols

As discussed in section 5.4, AFLNet cannot fuzz with a more expressive grammar, but only with a dictionary. A grammar that supports more complex protocol syntax could be evaluated. Replacing a dictionary with such a grammar, could help improve fuzzing performance. However, a more complex grammar is also more complex to create for new protocols. Therefore, benefits of such a grammar should be evaluated against the additional manual work involved.

6.4 Alternative fuzzing frameworks

In the literature, multiple fuzzing frameworks are currently used. Some fuzzing frameworks are introduced in chapter 3. An extensive comparison between fuzzers is difficult, since most fuzzers focus on specific use cases and comparisons are highly dependent on the chosen SUT. Additionally, fuzzing is an active research area where new techniques and frameworks introduced frequently. This thesis focuses on greybox fuzzing with AFLNet, which is an extension of AFL. Other extensions of AFL and more independent alternatives such as Fuzzowski[1] might be interesting to compare performance in specific environments. Also there exist whitebox fuzzers that use the instrumentation of AFL such as KleeFL[2].

6.5 Optimising fuzzing duration

For non-trivial programs, an exhaustive search trough the input space is infeasible. Therefore, at some point the fuzzer has to be stopped. At that point there could still be undiscovered vulnerabilities. To determine a suitable time to stop fuzzing, a Good-Turing frequency estimation might help [35]. The Good-Turing frequency might help to find the point where a fuzzer is saturated. A fuzzer is saturated if no new findings are expected to be found in a reasonable time frame. Suggestions how this could be implemented in AFL and AFLNet along with possible limitations of this method can be found online[3].

Specifically for stateful fuzzing it would be interesting if a Good-Turing frequency could also be calculated per state. This could provide an indication when a fuzzer should select a new target state.

[1] https://github.com/nccgroup/fuzzowski
[2] https://github.com/julieeen/kleefl
[3] https://bshastry.github.io/2018/10/08/good-turing-fuzzing.html

Chapter 7

Conclusions

Fuzzing is an automated approach to software security testing. As explained in chapter 2, there are several variants of fuzzing techniques. Fuzzing can be divided mainly in white-, black- and greybox fuzzing. Depending on the circumstances such as the specific target, available information and available fuzzing time, these techniques can outperform each other.

In the experimental setup of this thesis, the greybox fuzzer AFLNet is used. Background information regarding AFLNet can be found in section 3.1. AFLNet is a fuzzer that supports stateful fuzzing of network protocols. To fuzz stateful programs, AFLNet handles input as a sequence of messages instead of a single message. The initial input messages of AFLNet are also referred to as initial seed. This allows mutating only a target message instead of a random part of the input. Selective mutating helps preventing the fuzzer only reaching several initial states and never reaching deeper states.

AFLNet has been compared with other fuzzers in chapter 3. Based on this comparison, AFLNet has several advantages. For instance a regional mutation strategy results in a substantial reduction of time to find bugs compared to other stateful fuzzers. Also, AFLNet has a maintained code base with sufficient documentation to use and extend it. However, large parts of the code base are inherited from AFL. Functionality of AFLNet appears to be added without major refactoring of the new code base.

As the case study for this research, an implementation of an IRC server, NgIRCd is used as server under test (SUT). This server implements the messaging protocol IRC which is described in chapter 4. There are several versions of the IRC protocol. Ongoing, but not yet finalised, efforts for a new IRC version have already been partly implemented in the SUT. The existence of several versions which are mostly backward compatible makes the IRC protocol difficult to implement. Potentially, this can lead to vulnerabilities that would not occur with more concise protocol specifications.

In the experimental setup of chapter 5, several extensions of AFLNet have been evaluated. The focus of the experiments was on the client-to-server

communication of the IRC protocol. For this purpose an initial seed is used that consists only of client-to-server communication without any server-to-server communication. As baseline AFLNet has been extended to implement fuzzing of the IRC protocol. Furthermore, three approaches extending the baseline have been evaluated on the SUT:

- Dictionary based fuzzing;
- State-aware fuzzing;
- Fuzzing with checkpoints.

A short overview of the results for each approach can be found below. Some considerations regarding the conducted case study are discussed later in this conclusion.

Dictionary based fuzzing

AFLNet already supported fuzzing with dictionaries. However, for the IRC protocol no dictionary was previously available. Therefore, for this case study a dictionary has been created for the IRC protocol. This dictionary increases the number of paths found significantly compared to the baseline. This makes dictionary based fuzzing the most successful conducted experiment.

State-aware fuzzing

AFLNet supports stateful fuzzing. However, by default AFLNet has no notion which states of the SUT are fuzzed with a given input. Implementing state-awareness for the IRC protocol enables the SUT to target specific states. This could help ensure that all protocol states are covered when fuzzing. In the case study, state-aware fuzzing does not improve fuzzing performance significantly. Combining state-aware fuzzing with a dictionary results in lower performance. Also, the IRC protocol does not enforce specific command orders in most states. This could explain why the impact of targeting specific states is low.

Fuzzing with checkpoints

AFLNet handles input as a message sequence. To fuzz a “deep” state, a large part of this sequence will be repeated once for each tested input. A sequence in the experiments often contains around twenty messages. And, the execution speed is often less then one execution per second. So, to increase execution speed substantially, checkpointing could be used to restore parts of the input sequence.

The first attempt to apply checkpointing used a containerised SUT with Podman. However, this turned out not to work together with instrumentation injected by AFLNet. An alternative approach with DMTCP has been identified.

Limitations of results

All experiments have been repeated five times to obtain reliable results. Even with repetitions, the standard deviation for total paths found and maximum depth reached, did differ with a factor three between different experiments. The sample size is large enough to reason about substantial performance impacts. However, smaller differences might not be noticeable. For instance a larger sample size could show a significant improvement for state-aware fuzzing. Further research with a larger sample size is recommended.

Another consideration is that the fuzzing has been terminated after 500.000 executions per experiment. Some results might change if the number of executions would be increased. Based on the generated state machine in section 5.7.2 fuzzing had been stopped too soon to reach all states of the IRC protocol.

To estimate whether there are no bugs in the SUT or whether the scope of the experiments was too limited, a number of synthetic bugs has been introduced in the SUT. Several of these bugs have been found by the fuzzer. This suggests the scope of the conducted experiments is sufficient.

An approach to determine a reasonable time to stop fuzzing based on the already found results has been identified in section 6.5.

Bibliography

[1] Patrice Godefroid. "Fuzzing". In: *ACM, Communications of the ACM* 63.2 (Jan. 2020), pp. 70–76. DOI: 10.1145/3363824.

[2] 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*. ACM, Nov. 2019. DOI: 10.1145/3338502.3359762.

[3] Alex Barton. *ngIRCd*. Version 26.1. URL: https://github.com/ngircd/ngircd/.

[4] Christophe Kalt. *Internet Relay Chat: Architecture*. RFC 2810. 2000. URL: https://tools.ietf.org/html/rfc2810.

[5] Johannes Mayer and Christoph Schneckenburger. "An Empirical Analysis and Comparison of Random Testing Techniques". In: *Proceedings of the 2006 ACM/IEEE ISESE*. ISESE '06. ACM, 2006, pp. 105–114. DOI: 10.1145/1159733.1159751.

[6] T.Y. Chen, T.H. Tse, and Y.T. Yu. "Proportional sampling strategy: a compendium and some insights". In: *Elsevier, Journal of Systems and Software* 58.1 (Aug. 2001), pp. 65–81. DOI: 10.1016/s0164-1212(01)00028-0.

[7] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. "ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation". In: *57th ACM/IEEE DAC*. DAC '20. IEEE, 2020. DOI: 10.1109/DAC18072.2020.9218603.

[8] Marcel Böehme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and Reflections". In: *IEEE, IEEE Software* 38.3 (2021), pp. 79–86. DOI: 10.1109/MS.2020.3016773.

[9] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-Based Whitebox Fuzzing". In: *ACM, ACM SIGPLAN Notices* 43 (May 2008), pp. 206–215. DOI: 10.1145/1379022.1375607.

[10] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Jan. 2018. DOI: 10.1145/3243734.3243804.

[11] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "AFLNet: A Greybox Fuzzer for Network Protocols". In: *IEEE, IEEE 13th ICST*. 2020. DOI: 10.1109/ICST46399.2020.00062. URL: https://thuanpv.github.io/publications/AFLNet_ICST20.pdf.

[12] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *ACM, Communications of the ACM* 33 (1990). DOI: 10.1145/96267.96279.

[13] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. "Mutation-Based Fuzzing". In: *The Fuzzing Book*. Saarland University, 2019. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html (visited on 10/05/2021).

[14] Michal Zalewski. *American fuzzy lop*. Version 2.57b. URL: https://github.com/google/AFL.

[15] Patrice Godefroid. "Random Testing for Security". In: *Proceedings of the 2nd international workshop on Random testing*. ACM, 2007. DOI: 10.1145/1292414.1292416.

[16] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. *"Evolutionary Grammar-Based Fuzzing"*. 2020. arXiv: 2008.01150 [cs.SE].

[17] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems". In: *IEEE 22nd SANER*. IEEE, Mar. 2015. DOI: 10.1109/saner.2015.7081877.

[18] Bernhard Steffen, Falk Howar, and Maik Merten. "Introduction to Active Automata Learning from a Practical Perspective". In: *Springer, Formal Methods for Eternal Networked Software Systems*. 2011, pp. 256–296. DOI: 10.1007/978-3-642-21455-4_8.

[19] G. Harris and M. Richardson Sandelman. "PCAP Capture File Format". 2020. URL: https://pcapng.github.io/pcapng/draft-gharris-opsawg-pcap.html#name-references.

[20] *QEMU*. URL: https://www.qemu.org/.

[21] Van-Thuan Pham, Marcel Boehme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. "Smart Greybox Fuzzing". In: *IEEE, IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2019.2941681.

[22] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. "Fuzzing IoT Firmware via Multi-Stage Message Generation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 2525–2527. DOI: 10.1145/3319535.3363247.

[23] Rong Fan and Yaoyao Chang. "Machine Learning for Black-Box Fuzzing of Network Protocols". In: *Springer, Information and Communications Security*. Vol. 10631. 2018, pp. 621–632. DOI: 10.1007/978-3-319-89500-0_53.

[24] Christophe Kalt. *Internet Relay Chat: Channel Management*. RFC 2811. 2000. URL: https://tools.ietf.org/html/rfc2811.

[25] Christophe Kalt. *Internet Relay Chat: Client Protocol*. RFC 2812. 2000. URL: https://tools.ietf.org/html/rfc2812.

[26] Christophe Kalt. *Internet Relay Chat: Server Protocol*. RFC 2813. 2000. URL: https://tools.ietf.org/html/rfc2813.

[27] PlantUML Team. *PlantUML*. Version 1.2021.2. 2009. URL: plantuml.com.

[28] Roberto Natella and Van-Thuan Pham. *ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing*. 2021. arXiv: 2101.05102 [cs.CR].

[29] Supriya Bhalerao and Prashant Kadam. "Sample size calculation". In: *Medknow, International Journal of Ayurveda Research* 1 (2010). DOI: 10.4103/0974-7788.59946.

[30] S R Jones. "An introduction to power and sample size estimation". In: *BMJ, Emergency Medicine Journal* 20 (Sept. 2003). DOI: 10.1136/emj.20.5.453.

[31] Sakai Tetsuya. "Statistical Significance, Power, and Sample Sizes". In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2016, pp. 5–14. DOI: 10.1145/2911451.2911492.

[32] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. "Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools". In: *Springer, Graph Drawing Software*. 2004, pp. 127–148. DOI: 10.1007/978-3-642-18638-7_6.

[33] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In: *IEEE IPDPS*. IEEE. 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161063.

[34] Marco Henrix. "Performance improvement in automata learning". MA thesis. Radboud University, 2015.

[35] Marcel Böhme. "Assurance in Software Testing: A Roadmap". In: *IEEE/ACM ICSE-NIER*. IEEE, 2019, pp. 5–8. DOI: 10.1109/ICSE-NIER.2019.00010.

Appendix A

Appendix

A.1 Configuration of AFLNet

The installation instructions given on the AFLNet project site [1] are clear and help to get started quickly. Some general performance optimisations have to be activated at run time. For the case study, this has been done with the script given in listing 5.

```
# Set enviroment variables for AFLNet
export AFLNET=$(pwd)/aflnet
export WORKDIR=$(pwd)
export PATH=$AFLNET:$PATH
export AFL_PATH=$AFLNET

# Catch non crashing memory faults
export AFL_HARDEN=1

# Set the scheduler from the default 'ondemand' to 'performance'.
echo performance | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

# Disable huge pages
echo never > sudo /sys/kernen/mm/transparent_hugepage/enabled

# Set better scheduling strategies
echo 1 > sudo /proc/sys/kernel/sched_child_runs_first
echo 1 > sudo /proc/sys/kernel/sched_autogroup_enabled
```

Listing 5: A script to configure environment variables and performance settings for AFLNet.

[1] https://github.com/aflnet/aflnet

A.2 Podman checkpointing API

Podman can be used to checkpoint a container. Rootless containers are currently not supported with checkpointing. To enable fuzzing without root permissions on the fuzzer, the API can be used to interact with the containers. In that case the Podman service runs as root.

To test the API some functions are written in Python (3.9.4). This testing showed some inconsistent behaviour. According to the documentation [2] [3] it is possible to export a checkpoint to a tar.gz archive. This archive should be possible to import with a different name or on a different host. This could be used to fuzz multiple versions of SUT in parallel. However, when completing more than one full checkpoint-pause-restore cycle, the underlying service criu stops with error code 52. According to the log an invalid cgroup configuration occurs. This issue does not occur without exporting.

Hyperfine[4] is used to measure the required time for a full checkpoint-pause-restore cycle. These functions are exposed in a Python program shown in listing 6.

An initial state is created with a single container. For the benchmark 10 runs are measured. Additionally, 5 runs are executed directly before the benchmark that are discarded. According to this benchmark a full cycle requires on average 2.2 s±0.1.

A more representative measurement is creating a single checkpoint and restoring this multiple times. To simulate this behaviour, the same benchmark procedure as for the full cycle is used. A single checkpoint is created, and restored ten times. On average this takes 11.7 s (±0.3).

[2] http://docs.podman.io/en/latest/markdown/podman-container-checkpoint.1.html
[3] http://docs.podman.io/en/latest/markdown/podman-container-restore.1.html
[4] https://github.com/sharkdp/hyperfine

```
class PodmanBindings:

    def __init__(self, domain="localhost", port="8080", version="1.40.0"):
        self.url = f"http://{domain}:{port}/v{version}/libpod/"

    def get_containers(self):
        response = requests.get(self.url + "containers/json")
        response.raise_for_status()
        return response

    def stop_container(self, name, all=False, Ignore = False, timeout=10):
        request_url = self.url + "containers/" + name + "/stop"
        params = {"all":all, "Ignore":Ignore, "timeout":timeout,}
        response = requests.post(request_url, params = params )
        response.raise_for_status()
        return response


    def checkpoint_container(self, name, export=True, ignoreRootFS=False,
                             keep=True, leaveRunning=True, tcpEstablished=True):
        request_url = self.url + "containers/" + name + "/checkpoint"
        params = {"export":export, "ignoreRootFS":ignoreRootFS, "keep":keep,
                  "leaveRunning":leaveRunning, "tcpEstablished": tcpEstablished}
        response = requests.post(request_url, params = params, stream=True )
        response.raise_for_status()
        return response

    def restore_container(self, name,  keep=False, leaveRunning=True,
                          ignoreRootFS=False, podman_import=True,  ):
        request_url = self.url + "containers/" + name + "/restore"
        params = {"import":podman_import, "keep":keep,
                 }
        response = requests.post(request_url, params = params, )
        response.raise_for_status()
        return response
```

Listing 6: Bindings to use the Podman api to checkpointing and restore based on the json and request libraries.