

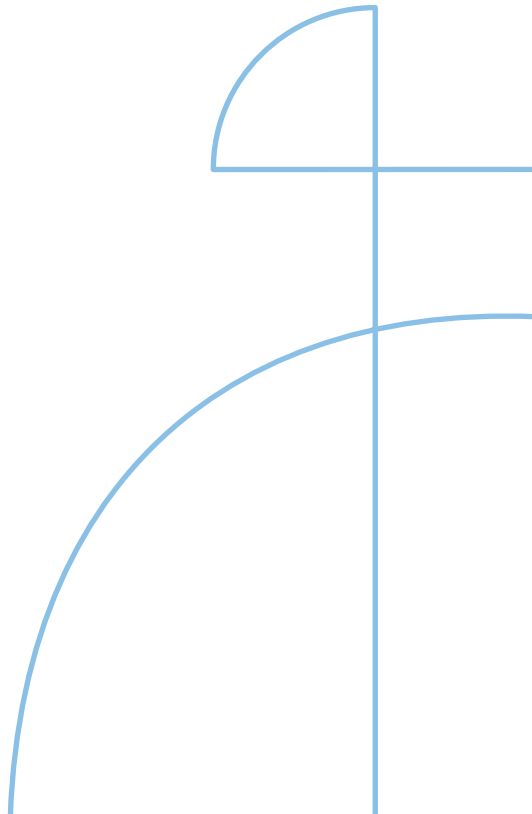


Licentiate Thesis in Computer Science

Black-Box Fuzz Testing for Security in Service-Provider Networks

LEON FERNANDEZ

KTH ROYAL INSTITUTE OF TECHNOLOGY



Black-Box Fuzz Testing for Security in Service-Provider Networks

LEON FERNANDEZ

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Licentiate of Engineering on Tuesday the 17th of March 2026, at 10:00 a.m. in D37, Lindstedtvägen 5, Stockholm.

Licentiate Thesis in Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2026

© Leon Fernandez

TRITA-EECS-AVL-2026:12
ISBN 978-91-8106-519-0

Printed by: Universitetservice US-AB, Sweden 2026

Abstract

Computer networks underpin many aspects of our daily lives. Familiar services such as digital payments, social networks, video streaming and messaging apps would not function without them. While the services we enjoy may seem stable on the surface, underneath the hood they are ever-changing: components are replaced, networks are rebuilt and source code is rewritten. Similarly, the threat posed by malicious actors is also in constant motion. What is considered secure today may not be secure tomorrow. This is especially true for software components. Therefore, software security testing is necessary to ensure that a service poses no risk to its operators nor its end-users.

A critical step in developing secure software is discovering previously unknown vulnerabilities. Fuzz testing, or fuzzing, is a state-of-the-art technique for preventing insecure software from being taken into production. One form of fuzz testing that has received great interest in recent years is grey-box fuzzing. Unfortunately, some systems are not well-suited for this type of testing. Implementation aspects such as programming language, statefulness, network connectivity and source-code availability can make grey-box fuzzing difficult. Consequently, not all types of vulnerabilities are discoverable with this technique.

In this thesis, I investigate a different approach to fuzzing: black-box fuzzing. As the name suggests, black-box fuzzing does not depend on implementation details about the target system. While this allows for testing a wider range of systems, it also pays a price by sacrificing speed and test coverage. However, if the black-box fuzzer can find vulnerabilities that a grey-box fuzzer cannot, it might be worth the price. The results I present in this thesis show that by incorporating elements from reinforcement learning and web crawling, black-box fuzzing can be used where grey-box fuzzing falls short to discover previously unknown vulnerabilities in real-world networking software.

Keywords

Cyber Security, Security Testing, Vulnerability Discovery, Fuzz Testing, Computer Networks, Network Protocols, Software Engineering

Sammanfattning

Datornätverk utgör grunden i många av våra vardagliga handlingar. Tjänster såsom digitala betalningar, sociala nätverk, strömmad video och direktmeddelanden är helt beroende av dem. Trots att tjänsterna vi nyttjar ger ett stabilt intryck befinner de sig i ständig förändring under huven: komponenter byts ut, nätverk förändras och källkod skrivs om. På samma sätt är hotet från illasinnade aktörer i ständig rörelse. Det som betraktas som säkert idag kanske inte är det imorgon. För mjukvarukomponenter är detta särskilt påtagligt och därför är säkerhetstestning av mjukvara nödvändigt för att en tjänst inte ska utgöra en risk för dess slutanvändare eller operatörer.

Ett kritiskt steg för att utveckla säker mjukvara är att upptäcka hittills okända sårbarheter. Fuzztestning, eller fuzzing, är den främsta teknik vi har idag för att förhindra att osäker mjukvara tas i produktionsdrift. En sorts fuzztestning som har krönts med stora framgångar under de senaste åren är grey-box fuzzing. Dessvärre lämpar sig vissa system dåligt för denna typ av testning. Implementationsaspekter såsom programspråk, tillståndsmodell, nätverkskonnektivitet och källkodens tillgänglighet kan försvåra grey-box fuzzing. Således kan vissa typer av sårbarheter inte upptäckas med denna teknik.

I denna avhandling undersöker jag en alternativ metod för fuzzing: black-box fuzzing. Som namnet antyder betraktar man med denna metod systemet som ska testas som en svart låda, en enhet vars implementation är okänd för oss som testare. Detta har fördelen att metoden kan användas för att testa en större bredd av system men man betalar ofta ett pris för detta i form av exekveringshastighet och testtäckning. Men om en black-box fuzzer hittar sårbarheter som en grey-box fuzzer missar så kan det vara värt priset. Resultaten som jag presenterar i denna avhandling visar att black-box fuzzing kan kombineras med förstärkningsinlärning och web crawling. På så sätt kan tekniken täcka upp för tillkortakommanden hos grey-box fuzzing och upptäcka tidigare okända sårbarheter i mjukvara för datornätverk.

Nyckelord

Cybersäkerhet, Säkerhetstestning, Sårbarhetsupptäckt, Fuzztestning, Datornätverk, Nätverksprotokoll, Mjukvaruteknik

Acknowledgment

First of all, I would like to thank my family. Thanks for keeping me grounded, motivated and for teaching me what security means in general (and not just for computers).

Secondly, I want to thank my supervisor, Gunnar Karlsson. Thanks for being such a patient and understanding mentor. I have met many great teachers throughout my life, but only one who has inspired me to never stop learning.

I would also like to thank my opponent, Juha Röning, my reviewer, Musard Balliu, and my examiner, Hamed Nemati. Thanks for setting aside some of your valuable time for this thesis.

Lastly, I would like to thank all my colleagues, both former and current, at NSE and elsewhere. Thank you for all the insights, laughs and cups of coffee that we have shared. It is a great privilege to work with people who are (in various combinations) kind, smart, good at finding lunch spots, funny, brilliant with numbers and thoughtful.

//

Leon
Stockholm, February 17, 2026

List of Included Papers

Papers Included in this Thesis

- Paper A** **Measuring the Impact of Fuzzing Activity in Networking Software - Extended**, Gunnar Karlsson and Leon Fernandez. In KTH Royal Institute of Technology Technical Reports, 2025
- Paper B** **Squashing Resource Exhaustion Bugs with Black-Box Fuzzing and Reinforcement Learning**, Leon Fernandez and Gunnar Karlsson. In 2023 7th International Conference on System Reliability and Safety (ICSRS), 2023
- Paper C** **Black-box Fuzzing for Security in Managed Networks: An Outline**, Leon Fernandez and Gunnar Karlsson. In IEEE Networking Letters, 2023
- Paper D** **Fuzz Testing for Code Injection Vulnerabilities in Network Management Systems**, Leon Fernandez and Gunnar Karlsson. In 2024 8th International Conference on System Reliability and Safety (ICSRS), 2024

Papers Not Included in this Thesis

- Paper E** **Measuring the Impact of Fuzzing Activity in Networking Software**, Leon Fernandez and Gunnar Karlsson. In Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, 2025

Contents

Abstract	iii
Sammanfattning	v
Acknowledgment	vii
List of Included Papers	ix
Contents	xi
1 Introduction	1
1.1 Type A and Type B Cybersecurity	1
1.2 The Testing Pyramid	2
1.3 Testing vs. Scanning	4
1.4 Vulnerability Discovery	4
2 Background	7
2.1 Black-box Fuzzing	8
2.2 White-box Fuzzing	9
2.3 Grey-box Fuzzing	10
2.4 Aspects in Focus	12
3 Producing Fuzzy Input	15
3.1 Generational Fuzzers	16
3.2 Mutational Fuzzers	19
3.3 Handling Statefulness	21
4 Failing a Test Case	23
4.1 Protocol Models	23
4.2 Code Injection	24
4.3 Resource Usage	25
4.4 Differential Fuzzing	26
5 Original Work	29
6 Conclusion	33
Bibliography	35

1 Introduction

What keeps you safe online? How is it that you are able to chat, share photos, book meetings, pay taxes and run businesses over the public Internet safely? As cybersecurity researchers we spend our days studying when things go wrong. Therefore, these questions might seem incorrect or misleading. But the truth is that, most of the time when someone goes online, nothing bad happens. And most of the systems out there in the public Internet, do not get breached. "Modern cryptography", someone might say, and with that they are correct, but it is only half the truth.

Cryptographic systems are theoretical constructs, existing perhaps as a PDF document in some technical organization's archive or in a thick course book. In order for them to actually protect you, they must be *implemented*. During the transfer process from written notes to a usable computer program, much can go wrong. How do we know that the programmer understood the PDF document correctly? How do we know that the programmer, despite having perfect understanding of what is to be implemented, did not make any mistakes while writing the code? The answer to this, the other half of the truth, is to test the program. And to test it again. And again. And again...

1.1 Type A and Type B Cybersecurity

Cybersecurity work can take many forms: mathematical analysis, software development, policymaking and popular education, to name a few. I like to make a distinction between two fundamental forms which I usually call "Type A" and "Type B", for lack

of better terms. Type A is the type of work I believe most people envision when they hear the word "cybersecurity". It is more related to being a *consumer* of digital technology. It encompasses tasks such as managing employee workstations at a company, setting up and monitoring an office network, establishing practices for handling sensitive data and informing colleagues about the hazards of clicking links in emails. Type B, on the other hand, is more related to being a *producer* of digital technology. This includes software development, system design, writing protocol specifications and, of course, testing.

As with any model, reality is always more complicated. Hence, there is no clear line between the Type A and Type B. Where would supply-chain security [6] fit in, for instance? Still, I believe it is important to make a distinction. While many decision makers today are willing to fund activities in the name of Type A, they are more reluctant to do so in the name of Type B. And this is reflected in the real world: when something bad happens to us online, it is often due to a piece of software failing to deliver on its promise of being secure. Perhaps a programmer made a mistake that turned out to be exploitable? Or perhaps the software is difficult to use securely because of unexpected side-effects and interactions with other software, thereby making human errors more likely? In other words, Type B cybersecurity was lacking. This thesis is about Type B cybersecurity.

1.2 The Testing Pyramid

The testing pyramid is prevalent in the software testing literature [7]. It comes in many different variants, but the core idea it conveys is always the same: simple tests are cheap and can be performed in large quantities while more qualitative tests take time and development effort and are therefore fewer and/or performed more seldomly. Figure 1.2.1 shows a simple testing pyramid with three levels of tests.

At the bottom level resides the bulk of the tests, typically unit tests. Unit tests are typically compiled into small programs that, when run, verify the behaviors of a selected part of the system. For example, that a function that sorts a list of strings in alphabetical order actually does its job and that the function's behavior stays the same even when the implementation changes. This is illustrated in Figure 1.2.1. Unit tests should be cheap and easy to run so that all developers can run them as often as they like.

In the middle reside the integration tests. In these tests, we

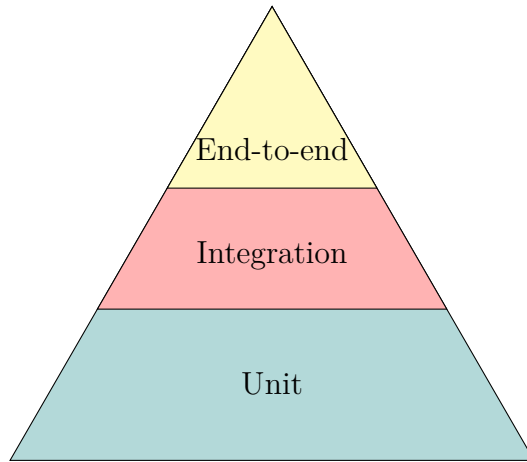


Figure 1.2.1: The testing pyramid.

might be testing a “properly compiled” binary, in contrast to unit tests. But we might test it in an environment that is simpler and more controlled than its intended production environment. For example, we might test a server program that we are writing using a dummy client program that uses a local connection. This allows us to verify a lot of behavior without having to care about difficult problems such as packet loss or failing connections. Still, maintaining a dummy client takes some additional work and the tests might take some time to run to completion, thereby limiting how many integration tests a project can be subject to.

At the top of the pyramid are the end-to-end tests, the most qualitative and time-consuming tests that we run against a system. They strive to be as realistic as possible by emulating usage in production. Setting these tests up might require significant efforts such as deploying a database, generating end-user traffic streams and configuring external network functions such as time or name servers. Depending on the system under test (SUT) and its development history, these tests may even be manual, which means that completing all end-to-end tests might take a long time even though they are few. It may also be difficult to run them in parallel since they might depend on scarce hardware resources such as traffic-generators or high-precision network clocks.

There is no clear distinction between a software bug and a software vulnerability. This even sparks heated debate sometimes [8, 9]. What appears to be a severe vulnerability might prove to be unexploitable in practice and seemingly harmless bugs can

cause exploitable behavior. Take, for instance, a web service that responds slightly differently to login attempts depending on whether the password was wrong or the username was non-existent. A harmless inconsistency some might say. But what if the service relies on phone numbers for usernames? An attacker can exploit the inconsistency to enumerate users of the service. Perhaps the inconsistency was not so harmless, after all? Because of the thin line between bug and vulnerability, all tests are to some extent security tests. Dedicated security tests can be more qualitative or quantitative in nature. Hence, security tests can exist at any level of the testing pyramid.

1.3 Testing vs. Scanning

Much of the work behind developing secure software consists of answering various forms of the question "Are we vulnerable against X?". There are many tools that operate in a point-and-shoot type of fashion when trying to answer this question. Some examples include `testssl.sh` [10], Zonemaster [11], Zed Attack Proxy [12] and Nessus [13]. You basically provide them with an IP address or a URL and they start examining the SUT for signs of *known* vulnerabilities. For example, `testssl.sh` looks for the usage of deprecated cryptographic algorithms during a TLS handshake. This type of activity is usually known as vulnerability scanning or security scanning. It can be understood as a form "security unit testing". That is, cheap bulk tests to flush out the most obvious vulnerabilities or potential vulnerabilities in the SUT.

While answering "Are we vulnerable against X?" is relatively straightforward, answering the open-ended version of the question, "What are we vulnerable against?", is much more difficult. Finding out whether a component of your system has a known vulnerability usually amounts to reading its release notes, checking against a vulnerability database or running an automated scanning tool. However, discovering *previously unknown* vulnerabilities typically require a much more qualitative form of testing. This is what we usually mean when we talk about security testing and it is a different type of activity than security scanning. So, without any known exploits to test, i.e. scan, our SUT with, how do we discover *new* vulnerabilities?

1.4 Vulnerability Discovery

Vulnerabilities tend to exist in the domains of the under-specified and the not-quite-understood: in the corner cases that have not

been thoroughly discussed or where the number of configuration options might be overwhelming. By definition, such cases can be difficult to formulate as tests or even to identify in the first place. Properly secured software must have its quirks and corner-cases sought out and rectified. Therefore, the testing process cannot rely on manually specified test cases exclusively.

Fuzz testing, or fuzzing, is a form of testing that to some extent uses randomness to automatically generate test case inputs. The underlying rationale is simple: if we cannot think of test cases for tricky corner-cases ourselves, we can at least hope to discover them by chance. Consequently, fuzz testing requires generating *a lot* of test cases. Despite this, fuzz testing is a highly qualitative process. Unlike unit tests, the generated test cases are typically not reused to ensure stability over time. They are instead a means that allow the tester to deeply explore the behaviour of the SUT in extreme or unusual conditions at a specific point in time during the SUT's development. There are numerous different fuzzing approaches, each with different pros and cons. Selecting and setting up the right one can be very time-consuming and may require substantial knowledge about the SUT. However, when it comes to discovering *new* vulnerabilities, fuzzing is the best tool we have.

2 Background

Despite the simple idea at the core of fuzzing — send random data to the SUT and monitor for crashes — there are many aspects to it. When comparing different approaches for fuzzing we typically place them into one of three categories: white-box fuzzing (WBF), grey-box fuzzing (GBF) and black-box fuzzing (BBF) [14]. They all follow the same general pattern:

1. Generate randomized input data for the test case.
2. Deliver it to the SUT via some mechanism.
3. Observe some part of the SUT's behavior.
4. Pass or fail the test case based on some condition for the observation.

Throughout this thesis, I will use terms like "random input data", "fuzzy data" and "test case" interchangeably, as in "a fuzzer might generate millions of test cases without exposing any bugs". The reason for this is that most fuzzers actually generate complete test cases that can be evaluated, not just the input data.

Another important term is "campaign". When someone starts a fuzzer, lets it repeat steps 1-4 above, stops it and then analyses the results, we call it a "campaign". This is an important term because, as we shall see later, modern fuzzers can adapt their behavior gradually during a campaign to obtain better results.

2.1 Black-box Fuzzing

As the name implies, BBF assumes no prior knowledge about the internals of the SUT. It is agnostic to implementation details of the SUT such as programming language, execution environment and whether runs in a distributed fashion or not. A BBF just needs two things to function:

- A way of delivering fuzzy data for the SUT to act upon.
- A way to check whether the SUT has been brought to an undesirable state.

Many BBFs act as a sort of "faulty end-user" that inputs corrupted data to the SUT via e.g. network packets or command-line arguments and then checks the SUT's health in order to pass or fail a test. The original paper that coined the term "fuzz" makes a clear example [15]:

- Send random characters through a UNIX pipe to the SUT.
- Check whether the SUT terminated abnormally (crashed) or hung (did not terminate within five minutes).

Another approach might be:

- Send a malformed HTTP request to the SUT (i.e. a web server).
- Check whether the response contained a 5xx status code (indicating a server fault).

By virtue of being implementation-agnostic, BBF is widely applicable and can be applied to SUTs of arbitrary complexity. Because they can be applied to production-grade SUTs, they also do not suffer from false positives that can arise when modifying the SUT (such as by stubbing out parts of its code to make testing easier). BBF also has some drawbacks. Namely, test case execution can be much slower (especially for networked SUTs) and it can be more difficult to observe undesirable behavior such as memory leaks or crashing processes.

To somewhat mitigate the latter drawback, some light instrumentation of the SUT is usually acceptable within the BBF approach; we are not entirely limited to response codes or exit statuses. Many systems have support for features such as sending logs to an external system or remote performance monitoring. We can of course use that as a basis when evaluating a test case.

BBF is often viewed as a fallback measure to use when one’s preferred fuzzing method cannot be applied. For that reason, BBF has received less attention from academia in recent years while GBF has seen ever-increasing heights of popularity. Somewhat contradictory, it is also widely recognized that different fuzzers find different types of vulnerabilities. As we shall see in later chapters, BBF’s agnosticism to implementation details lets it find unique vulnerabilities and that there are indeed scenarios where we should keep it in our belt rather than at the bottom of our toolbox.

2.2 White-box Fuzzing

In BBF we have little to no information about the implementation details of the SUT. White-box fuzzing (WBF) refers to the opposite scenario: we know everything about the SUT implementation. We might for instance have access to its source-code, or some intermediate representation thereof, that we use to analyse and reason about the program behaviour without actually running it. This naming is derived from the more general notions of black-box and white-box testing. The term WBF was coined by Godefroid in 2007 [16] to highlight a novel type of fuzzer that could use insights gained from processing the SUT’s internals to generate better test cases. Note that there is no clear definition of “better test cases” in fuzzing. But if fuzzer A causes 80% of the lines of code in a SUT to be executed while fuzzer B reaches only 60% within the same amount of test cases, then we would typically say that the fuzzer A generated “better test cases”. Other factors may weigh in too, of course. Execution time and the severity of the findings also matter when evaluating a fuzzer’s input generation mechanism. But achieving a high degree of coverage is nonetheless important. After all, code that remains unexecuted also remain untested.

Consider the code snippet in Figure 2.2.1. A BBF that generates completely random input that gets interpreted as the `Packet` type has a very low probability of hitting the `process_msg` and `process_other_msg` functions (less than 1 in two billion of hitting either). This makes BBF inefficient since we are not fuzzing any interesting program logic most of the time. This is where white-box fuzzing can help. By inspecting the internals of the SUT, which Figure 2.2.1 is assumed to be a part of, a WBF can overcome these inefficiencies. By employing techniques such as symbolic execution, tools like SAGE [17], KLEE [18] and Driller [19] can impose conditions on the generated input by solv-

ing constraints that block certain execution paths, like lines 14 and 17 in Figure 2.2.1. They can also use taint tracking techniques to reason about where different parts of the input data flow. Taint tracking refers to techniques that analyse how untrusted (tainted) data affects the behavior of the SUT without actually executing the code. For example, if input data is read from a network socket into an array variable, that variable may be considered tainted. At a subsequent location in the code, the first half of that tainted array is used in some important operation such as a database query or a system call. WBFs can use that knowledge to their advantage by fuzzing selected parts of the input more aggressively. In this case that would be the first half of the input data. Tools like VUzzer [20] and Angora [21] employ such strategies.

2.3 Grey-box Fuzzing

While WBF can significantly improve the quality of the generated test cases, it also comes at a significant computational cost. The employed techniques, symbolic execution, taint analysis, constraint solving and the likes, all face scalability issues in large codebases. This can make WBF too slow to be of practical use in some scenarios. In other words, there is a trade-off to be made: we can trade the granularity of our knowledge of the SUT internals for test case execution speed. Fuzzers that leverage this trade-off are usually referred to as grey-box fuzzers. By solely relying on coverage reports from the executed test cases, GBFs can in some sense "learn" how the features of the generated input affects how the SUT executes. This is much more lightweight since the expensive static-analysis of the source-code is done away with. Consequently, a GBF can run many more test cases within a given amount of time in an attempt to execute difficult-to-reach parts of the SUT by sheer volume.

A simplified algorithm for a generic GBF is shown in Figure 2.3.2. It starts with a number of initial seeds containing the type of data that the SUT expects. In practice this could be PDF files, JPEG images, captured network packets or URL strings, for example. Every iteration of the campaign (in the `while`-loop) starts by picking a seed. From it, the GBF generates the fuzzy input, more on this in chapter 3.2. Then, the GBF executes the SUT with each fuzzy input. If the execution yields any "interesting" discoveries, such as new paths being executed, the fuzz that was used is stored as a seed for further mutation in a subsequent iteration. A history is also kept for each seed based on how its derived fuzz performs. Once all seeds have been fuzzed, some

```

1 typedef struct Packet {
2     int msg_type;
3     int size;
4     char payload[];
5 } Packet;
6
7 int handle_msg(Packet *pkt) {
8     int retval = 0;
9
10    if (NULL == pkt) {
11        return -1;
12    }
13
14    if (pkt->msg_type == 1000) {
15        retval = process_msg(pkt->size, pkt->payload);
16    }
17    else if (pkt->msg_type == 2000) {
18        retval = process_other(pkt->size, pkt->payload);
19    }
20    else {
21        printf("Invalid message type!\n");
22        retval = -1;
23    }
24
25    return retval;
26 }
27
28 int process_msg(int size, char *payload) {
29     /* Complex processing */
30 }
31
32 int process_other(int size, char *payload) {
33     /* Also complex processing */
34 }

```

Figure 2.2.1: A snippet of code with which BBFs might struggle to reach any complex processing logic, which is typically what we want to fuzz in the first place.

```

1 seeds = {"aaa", "bbb"}; /* initial seeds */
2
3 int main() {
4     while (length(seeds) > 0) {
5         seed = get_next(seeds);
6
7         fuzzy_inputs = generate_fuzz(seed);
8
9         for fuzz in fuzzy_inputs {
10            exec_info = execute_SUT(fuzz);
11
12            if new_discoveries(exec_info) {
13                append(seeds, fuzz)
14            }
15
16            update_history(seed, exec_info)
17        }
18
19        mark_fuzzed(seed)
20
21        if all_seeds_fuzzed(seeds) {
22            for seed in seeds {
23                history = get_history(seed)
24                discard_or_keep(seed, history)
25            }
26
27            unmark_all(seeds)
28        }
29    }
30 }

```

Figure 2.3.2: Pseudo-code for a generic grey-box fuzzer.

seeds are discarded based on their history. Maybe they have not yielded any new discoveries in a long time or a new seeds have been found that covers the same code but executes faster. Once this "culling" is done, the GBF begins a new cycle where it fuzzes the culled list of seeds.

2.4 Aspects in Focus

There are many aspects of fuzzers to study, many of them orthogonal to whether a fuzzer is black, white or grey. For instance, a fuzzer might use metadata from failures to deduce whether two failed testcases are due to the same underlying root cause, as in [22, 23]. A fuzzer might be initialized with pre-recorded input data, the choice of which can affect the outcome of a cam-

paign [24, 25]. Several testbenches have been devised to facilitate the comparison of fuzzers [26, 27, 28, 29] and their value as benchmarks have been studied [30, 31]. Implementation details of fuzzers and their consequences have been studied [32, 30] as well as fuzzer usage "in the wild" amongst practitioners [33].

The focus of this thesis is mainly on black-box fuzzing for networked systems. More specifically, we focus on two fundamental aspects thereof: generating fuzzy input and detecting failures. These aspects will be investigated in chapters 3 and 4, respectively. Chapter 5 presents this thesis' contribution to the understanding of black-box fuzzing.

3 Producing Fuzzy Input

The most important aspect of a fuzzer is its means of input generation. After all, automatic input generation is what distinguishes fuzzing from other forms of software testing. A fuzzer should be able to generate input that is malformed in such a way that it passes the initial stages of validation in the system under test (SUT) and reaches deeper parts of the system. There, it is more likely to succeed in its task of finding possible security vulnerabilities. Two broad categories of input generation exist and are in use: generational and mutational. Generational fuzzing is also sometimes referred to as model-based or grammar-based fuzzing. Generational fuzzers, use a model that encodes the general structure of the input that the SUT expects. The model can then be used to programmatically generate fuzz that looks like what the SUT expects. Mutational fuzzers, on the other hand, generate fuzz by taking pre-recorded input, the seeds, and altering it by choosing from a set of mutation operations that it applies to the input seed.

For black-box fuzzers (BBF), input generation is especially important. By definition, the input is the only means with which a BBF can explore the SUT. Without coverage information or source-code inspection, there is not much else a BBF can do to reach deep parts of the system other than generating quality input from the start. Luckily, for many networked applications, the input space is fairly well-delimited in the form of a protocol specification. Inputs adhering to such a specification are likely to reach deeper parts of the SUT without as much need to "discover" interesting input features. This is opposed to data formats

such as PDF [34] or DOCX, which have many more degrees of freedom.

It should be noted that classifying fuzzers as either purely mutational or purely generational gives an over-simplified view of the reality and most successful fuzzers employ both strategies to some extent.

3.1 Generational Fuzzers

The seminal work by Miller et al. [15] describes a generational fuzzer. Input was generated programmatically using a very simple input model: a random-length sequence of random bytes. The model proved effective, even in follow-up studies done several years later. However, it was noted by security researchers that a completely random model mainly found shallow bugs [35, 36]. That is, errors in the initial stages of input validation and not in the actual business logic of the SUT. To tackle this, Aitel introduced in 2001 his fuzzer SPIKE. SPIKE models the input as a sequence of blocks. Each block consists of a number of bytes and can have certain features and/or relationships to other blocks. A block's features and relationships determine how it is used to generate fuzzy input. Other fuzzers have since adopted the block-based approach of input modeling. Most notably Sulley [37], its successor Boofuzz [38] and Peach [39].

Algorithm 1 describes a protocol data unit (PDU) consisting of a type-length-value (TLV) tuple, a common pattern used in many network protocols. The block representing the type-field describes a field that is one byte long, has a value of "7" and a "static" property. This means that this block will not be altered, a useful block property if we want to fuzz a specific message type. Next, there is a "size" block which contains two bytes and is guaranteed to properly encode the length, of the subsequent block. Fuzzy inputs with faulty length fields can be interesting, but they tend to trigger shallow bugs, which is something that SPIKE can avoid. Last follows the actual data value of the tuple. It consists of two blocks: a string block which will be altered in a mutation-style fashion and a random-length block of randomized bytes. This way, the block-style approach manages to avoid randomness where it is not desired, e.g. in fields that are expected to encode critical features of the PDU, and introduce it where it is desired, e.g. in fields that contain data for the business logic of the SUT.

Another way of modeling the input is with a grammar. Gram-

Algorithm 1: Snippet highlighting the concept behind block-based descriptions.

```

// Block description of a generic TLV starts here
s_start("generic_TLV");
// A single-byte type field that will not be altered
s_byte(7, static=True);
// A two-byte length field based on data block
s_size("generic_TLV_data", 2);
// Block description for the data block starts here
s_start("generic_TLV_data");
// First few bytes in data block is a string
s_string("Hello, world!");
// Then follows 0 to 100 bytes of random data
s_random(0, 100);
// End the inner data block
s_end("generic_TLV_data");
// End the outer TLV block
s_end("generic_TLV");

```

grammars have been used for a long time to describe several commonly used communication protocols [40]. There are many different kinds of grammars, and many different ways to represent them. Fuzzers have mainly followed the trend set by protocol designers of using context-free grammars (CFG) represented in Backus-Naur form (BNF). An example of such a description can be seen in Figure 3.1.1. The model is similar in many ways to the one in Algorithm 1, with some notable differences. Most notably, we cannot express the "length"-field with a CFG. Instead, we rely on an annotation (shown in blue) to do post-processing in order to "repair" broken parts of the generated input. The method of annotations, described in detail in [41], can be used to implement other complex dependencies between grammar elements, such as checksums, and also to express the probability of a certain grammar element being used during the generation phase. Another difference is the fact that we cannot easily express mutation operations with CFG. In the example in Figure 3.1.1 we rely on recursive list definitions in the grammar to produce fuzzy data similar to what we would get from Algorithm 1.

A milestone in the early days of black-box fuzzing is the PRO-TOS [35] project. Although it uses the term "fault injection" to describe its testing methodology, in modern terminology it is most accurately described as a grammar-based black-box fuzzer.

```

    <start> ::= <type><length><value>
    <type>  ::= 7
    <length> ::= <byte><byte>post=fix_length
    <value>  ::= <string-field><blob-field>
    <value>  ::= Hello, World!
              | <alphanum-garbage>
    <blob-field> ::= <random-garbage>
    <alphanum-garbage> ::= <alphanum-garbage><alphanum> | <alphanum>
    <random-garbage> ::= <random-garbage><byte> | <byte>
    <alphanum> ::= 'a' | 'b' | 'c' | ... | '8' | '9'
    <byte>     ::= 0x00 | 0x01 | 0x02 | ... | 0xFE | 0xFF

```

Figure 3.1.1: A BNF grammar that describes an input format similar to that in Algorithm 1.

It has since evolved into the proprietary commercial projects Codenomicon and later, Defensics.

Another fuzzer relying on CFGs is Grammarinator [42]. Grammarinator generates fuzzy input using a grammar specified in ANTLR notation [43], a widely used specification format for grammars. ANTLR has also been used in other fuzzers such as Nautilus [44].

Many networked applications are built using an interface description language (IDL) such as protobuf [45] or OpenAPI [46]. An IDL can help the developer to automate tasks such as documenting a service and writing boilerplate code. The developer writes IDL code that describes the structure that incoming data should conform to, and the structure of the data that will be sent in return. Some fuzzers use IDL specifications as input models. This makes them easy to use in a black-box fashion against SUTs that have been developed using popular IDL. RESTler [47] and EvoMaster [48] fall into this category.

In [49] the authors took a novel approach by modeling the generation of malicious HTTP requests as a reinforcement learning (RL) problem. The goal was to find web applications where code could be injected into the pages served by the app, a so-called cross-site scripting attack. Two RL agents collaborated to achieve this. One generated strings containing snippets of HTML

and JavaScript. If such strings are part of an HTTP response, there is a risk that a browser trying to render the response misinterprets them and embeds them in a page. The other changed the encoding of those strings to make sanitization more difficult. This included actions such as introducing percent-encoding, commonly used to encode certain characters in URLs, or changing the casing of the string.

3.2 Mutational Fuzzers

Mutation-based fuzzers produce fuzzy input by modifying existing input. They do so by relying on a set of mutation operations. These mutation operations are largely based on heuristics about common programming mistakes [50]. Off-by-one errors, wrong endianness or wrong signedness are examples of such mistakes that have inspired mutation operations. Depending on the SUT, the operations can also be more domain-specific, such as bad UTF-encoding or altering string casing. Operations can also be enhanced by dictionaries or generational methods that are used to replace or inject more complex data into the input [51, 44, 52]. Moreover, fuzzers such as VUzzer [20] and AFL and its derivatives [53, 54, 55, 56, 32], also employ so-called evolutionary strategies, which involve the splitting and combining of multiple inputs to produce fuzzy input.

AFL has been highly influential in its approach to mutation-based fuzzing. As described by [30] and [32] it employs the following *deterministic* strategies:

- Iterate over all bits and produce fuzzy input where each bit is flipped.
- Do the same for pairs of adjacent bits, triples of adjacent bits up to 32-tuples of adjacent bits.
- Iterate over all bytes and produce fuzzy input where each byte has had certain constants added to it.
- Iterate over all bytes and replace them with certain constants such as 0, 1 and -1.
- Do the same for pairs of adjacent bytes and 4-tuples of adjacent bytes.
- Iterate over a list of user-defined tokens and insert those at certain offsets.

AFL also employs two *random* strategies. Somewhat simplified, they consist of:

- Applying a random number of mutations, chosen randomly from the previous stages, to the same input
- Splitting two inputs at a random location near the middle and cross-combining them

It is difficult to understate the impact AFL has had in the fuzzing community. Consequently, much of the research in recent years has been focused on mutational grey-box fuzzing (GBF). Mutational strategies for BBF are less common. The reason for this is the fact that BBFs must produce quality input from the outset in order to be as efficient as possible, something that a generational strategy can guarantee. Despite this, there are some notable exceptions of mutational black-box fuzzers.

In [57] the authors use a scheme somewhat similar to that of AFL’s deterministic strategies. By starting off with byte-by-byte mutations and by analyzing the responses (rather than the GBF coverage data), Snipuzz is able to roughly map which byte groups (“snippets”) cause different parts of the code to execute. This is somewhat similar to the knowledge a grey-box fuzzer obtains and which makes them so efficient.

Focusing on SSL/TLS implementations, Frankencerts [51] received attention by addressing the challenge of generating highly structured input. For the Frankencerts use case it was X.509 certificates. The authors’ approach was to scan the web for unique certificates to build up a seed corpus. These seed certificates are then broken down into their constituent parts and randomly recombined in a mutational fashion. Furthermore X.509v3 supports customized information fields known as extensions, which Frankencerts can generate from a model and attach as a final production step. From a seed corpus of roughly 240,000 unique certificates, approximately 8 million syntactically valid (but semantically nonsensical) “frankencerts” were produced and used to test a number of SSL/TLS client implementations and web browsers.

Distinguishing between generational and mutational fuzzers can sometimes be difficult, as in the example of PULSAR [58]. Given the definition of a mutational fuzzer at the start of this section, it is a mutational fuzzer since it operates on pre-recorded input in the form of network packet captures. However, it does not directly mutate and replay the packets. Instead, it uses tech-

niques from [59] to vectorize and cluster the network packets and subsequently identify distinguishing features and events in the traffic. This information is then used to model the input and the protocol state and can be use both to check for "correctness" (conformance to the behavior seen in the pre-recorded data) and generate fuzzy input. The main drawback of this approach is that the generated fuzz can only contain features seen in the pre-recorded input. Protocol features that are absent from the pre-recorded set will not be fuzzed.

3.3 Handling Statefulness

For a fuzzer to ensure good test coverage, it must be able to take into account the possible statefulness of the SUT. Test cases comprising of a single transmitted message might not be enough to thoroughly exercise the SUT, no matter how cleverly that message has been fuzzed. A fuzzer might need to exchange a number of well-formed messages in order to bring the SUT to a specific state before sending a fuzzed message.

Sulley and Boofuzz pose a simplistic solution to the problem by introducing what they call a session graph, which is specified by the tester. It is a directed acyclic graph where each node is associated with a block-based description of a particular message. The fuzzer starts at the root node and fuzzes the corresponding message. After a certain amount of time, the fuzzer moves on to the next node and fuzzes its message, but begins each transmission with a well-formed version of the preceding message(s). After the entire graph has been traversed this way, the fuzzing campaign has run to completion.

Peach and PROTOS follow in a similar vein, with the tester manually specifying which message exchanges should take place and at what point the fuzz should be applied. In contrast to the session-graph, Peach's and PROTOS' state models can have awareness of the incoming messages and report on deviations.

State modeling can also be automated, as shown by Enemy of the State [60] and PULSAR [58], albeit not without a cost. Enemy of the State pays the cost of generality. It strives to emulate the behavior of a human using a web browser, which lets it infer the state on-the-fly. However, this makes it heavily reliant on web-specific concepts such as HTML elements, the Document Object Model, and HTTP methods and response codes. PULSAR pays the cost of being limited to test functionality that is present in the training data used to generate the model. Interesting to note,

though, is PULSAR’s ability to model both client and server, and its ability to act as a fuzzer in either role.

Building on previous research to apply GBF for networked systems [61, 26], StateAFL [62] augments it by taking into account the state of the SUT. It also models the state automatically by inferring by inspecting the process memory of the SUT during the campaign. If memory that tends to be constant across several request-reply roundtrips changes, that is indicative of a state change. Other fuzzers using similar techniques of tracking variables that are used to hold the state of the SUT [63, 64].

In [65], the authors noted that for web applications, the state is typically held in a database. By injecting the generated input directly into the database, the SUT is immediately forced to a specific state that might have been difficult to reach by a series of request-response exchanges. A scanner then scans the generated web pages for signs of code injection. Any findings are reported as potential risks since data should not pass from the database to the web front-end unsanitized.

In [66], the authors took a radical approach to handling statefulness. Rather than writing a fuzzer that sends corrupted input, they modify existing applications to act in unexpected ways. For instance, they fuzz a server by taking a corresponding client application and modifying the data it handles as well as its control flow. They call the modified application a “weird peer”. By being an *almost* correct client with *almost* correct state handling, the SUT is forced to handle a wide variety of situations, thereby achieving high coverage. Their approach also handles the inverse scenario, i.e. fuzzing a client by letting a server be the “weird peer”.

4 Failing a Test Case

Detecting failures is yet another fundamental property of a fuzzer. Defining what constitutes a failure can be difficult, however. It is the single most important factor that determines what types of vulnerabilities the fuzzer can find. Getting it wrong can result in no findings or too many (false positive) findings. The traditional approach to failure detection, that is still widely used today, was the approach taken by Miller et al. in their seminal fuzzing paper [15]. What their fuzzer looked for was crashes, typically caused by segmentation violation errors, or hangs, caused by the SUT being stuck in a loop or a deadlock. These are still the main means of failing test cases for modern fuzzers such as AFL and libFuzzer [67]. It is also the main means of crash discovery for fuzzers that connect to the SUT wirelessly, such as Snipuzz [57] and SweynTooth [68].

This approach has its limitations. Not all vulnerabilities manifest themselves as an unrecoverable crash, especially not for memory-safe languages such as Python or Go. For that reason, substantial research efforts have been made to increase the vulnerability diversity in fuzzing.

4.1 Protocol Models

For SUTs adhering to a network protocol, a natural way of failing test cases is to use fault codes within the protocol or to detect deviations from the protocol. Using fault codes is fairly straightforward, response codes are typically well-defined and most protocols support some kind of "server fail" signal. Many network

fuzzers like SweynTooth [68], AFLNET [61], EvoMaster [48] and RESTler [47] use this approach. This strategy can be further enhanced if there is a more elaborate model available. Many networked systems, especially RESTful services, are based on an IDL that models what response features can be expected for the supported requests. Deviations from this "IDL-based" model are used to fail test cases by fuzzers like EvoMaster and RESTler.

Writing models can be difficult and time-consuming. For that reason, efforts have been made to automatically "learn" a model, which can then be used to check for inconsistent behavior. This topic is closely related to that of handling state during fuzz generation, discussed in Section 3.3. Note that the purpose here is different: failing test cases based on deviant behavior versus producing input that reaches deep parts of the SUT.

4.2 Code Injection

Many vulnerabilities manifest themselves as code being injected and executed by the target system without causing the system to crash. This is especially common in web applications and can happen in different ways. Naturally, fuzzing research has attempted to address this. Since code injection can happen in a number of different ways, different techniques are required to detect whether a fuzzed input has caused an injection.

In [69], Appelt et al. tested various web applications for SQL injection vulnerabilities. Their approach involved a proxy in front of the database. The proxy was trained to learn the access patterns under "normal operations" as an initialization phase. During the testing phase, the proxy would then be used to determine whether the generated input had managed to alter the structure of the SQL commands or not. A different approach was taken by Trickel et al. in [70]. They reasoned that under normal operations, all SQL statements ought to be well-formed and not raise any database errors. Thus, a database error is an indication that a generated test case has managed to alter the structure of an issued SQL statement, which should not be possible. By instrumenting the SUT to detect errors in the responses from the database, they could fail test cases on the grounds that altering the structure of SQL statements should not be possible.

Web applications regularly make calls to the operating system. They might remove an old log file, for instance, or make a server-side request using a non-web protocol such as SNMP. If a part

of a request has an influence on the command string issued to the operating system, an attacker may be able to issue arbitrary OS commands via HTTP requests. In [70], Tricker et al. did not limit themselves to detecting SQL injection vulnerabilities. Using a similar reasoning, they instrument the SUT to fail test cases if there was a parsing error in the SUT's command interpreter as this should not happen under normal circumstances. Another approach was taken by Güler et al. in [71], where they placed a custom binary on the SUT that, upon invocation, signals to the fuzzer to fail the test case. By using a specialized mutation dictionary designed to trigger their custom binary, they could detect successful command injections and subsequently fail the test case.

A third injection vulnerability commonly found in web applications is what is known as a cross-site scripting (XSS) attack. An XSS allows an attacker to manipulate part of the webserver's response in such a way that malicious code can be embedded therein. It differs slightly from the previous two types of injections because the effects of the attack is not easily discernible at the server, but at a client (a web browser) that views the response. Naturally, this affects how fuzzers detect XSS vulnerabilities. For instance, the HTML in the response can be parsed into a tree, which can then be analysed to see if the attacker's code has been embedded. This approach was used by Duchene et al. in [72]. Eriksson et al. went with a different approach in [73], similar to that of Güler [71], where they injected a JavaScript function in all responses. The fuzzer would then try to embed a call to this function, where a successful call indicates a successful injection so that the test can be failed.

4.3 Resource Usage

All programs have to manage resources. Be they general resources like CPU cycles and memory, or application-specific resources like network nodes. Improper use of resources is often very difficult to discover. Oftentimes it is only detected after a long period of continuous uptime. Fuzzing can attempt to speed up this process by tracking resource usage throughout a campaign.

Memory is a critical resource for any program and memory mismanagement can have widely varying consequences, especially in memory-unsafe programs like C and C++. A common consequence is a hard crash due to a segmentation fault, the prime target of modern fuzzers like AFL and libFuzzer. But the consequences can also be more subtle.

Uncontrolled memory usage can cause denial-of-service (DoS) attacks by exhausting the memory of the SUT. This is a flaw commonly found in C and C++ programs as a result of not freeing memory that has been dynamically allocated by calls to `malloc()` or similar. To detect such flaws, Wen et al. used AddressSanitizer to detect memory leaks and uncontrolled memory usage, although not in a black-box fashion.

CPU cycle exhaustion is another form of DoS attack that aims to trigger worst-case execution paths in the SUT, thereby making it spend an excessive amount of CPU cycles on processing a specific input. It is also known as an algorithmic complexity attack. In [74] introduced SlowFuzz, a GBF that stood out by not prioritizing seeds that executed quickly or covered many lines of code, like AFL and libFuzzer, but by prioritizing seeds that executed slowly. By doing so, it could discover inputs that would cause significant slowdowns in several popular programs. Their evaluation strategy consisted of running the campaign for a fixed number of iterations and then looking at the worst-case execution times to identify inputs that cause performance problems. A similar approach was taken by Lemieux et al. in [75]. Their fuzzer, also a GBF, strived to be a bit more general by providing a framework for using arbitrary performance metrics to complement the traditional grey-box approach of using code coverage. However, in their evaluation they fell back on similar metrics as employed by Petsios (long execution time).

Many resource usage-driven fuzzers are GBF or WBF. This follows naturally from the relatively high level of instrumentation required to track resource usage. A notable example is [76], a continuation of the work on RESTler [47]. There they introduce a number of rules, based on heuristics, that may not be violated during the fuzzing campaign. REST, being HTTP based, use regular HTTP methods such as GET, POST and DELETE. The rules help enforce the semantics of these methods. For instance, a request to retrieve user information in a social network, `GET /users/leon`, should always return a HTTP 404 if preceded by a deletion request, `DELETE /users/leon`. Otherwise, a "user resource" is leaked. Any violation of these rules causes the test case to be failed.

4.4 Differential Fuzzing

Sometimes, there is no clear way to determine whether a network response is right or wrong, especially when responding to input that strives to be corrupted. But if a protocol is being followed,

then ideally responses should be the same across different implementations. If two different implementations respond differently to the same test case, then there is a good chance that at least one of them is wrong. Fuzzers that fuzz multiple SUTs in parallel and fails test cases when there is disagreement in the responses are called differential fuzzers.

Transport layer security (TLS) is a ubiquitous protocol. Consequently, there are many different implementations. Brubaker et al. [51] tested 8 libraries and 4 browsers on malformed X.509 certificates and found a significant amount of disagreement between the implementation. They found disagreement both in which certificate and certificate chains were accepted and how rejection was communicated to the end-user.

The domain name system (DNS) protocol is another ubiquitous protocol with several notable implementations. Bushart et al. used a differential approach for their DNS-specialized GBF [77]. They used a number of metrics to observe different behaviour such as the response contents, any server-side queries sent by the SUT and the state of the cached DNS records. However, they concluded that metrics other than coverage feedback and the response contents were difficult to utilize. In [78], Zhang et al. managed to utilize cache information to detect cache-poisoning attacks. By vectorizing and clustering the differences in the cache contents of six different DNS resolvers fuzzed in parallel, they were able to find anomalies that, after manual investigation, were found to be exploitable.

5 Original Work

The work in this thesis was motivated by my industry work experience as a software developer within the telecommunications sector. I wanted to learn more about security testing but found the current dominating paradigm, grey-box fuzzing, difficult to apply to the type of systems and development environments I was used to. Therefore, I set out learn more about alternative approaches to fuzzing and to contribute to the corresponding research. Many different questions arose during this process, but the core ones are:

- Q1** What are the shortcomings of grey-box fuzzing when it comes to networking software?
- Q2** How can formalization help the understanding of black-box fuzzing?
- Q3** How can non-intrusive system monitoring improve the input generation strategy of a black-box fuzzer?
- Q4** What vulnerabilities arise from the integration of different components in a distributed system?
- Q5** How can black-box fuzzing be applied to distributed systems?

All included papers in this thesis were co-authored with my supervisor, but the work behind answering the questions above was done by me. He provided very valuable feedback and advice, but I believe I am fair when I am claiming the contributions of the papers for my thesis.

In Paper A I wanted to find out how grey-box fuzzing was helping people working with networked systems. My personal experience was that it was very difficult to get a grey-box fuzzer up-and-running for the system I was working with at the time. In an effort to learn more about what I could gain or lose by choosing a different approach, I turned to Google’s OSS-Fuzz project [79]. Many open-source projects have submitted fuzzing configurations to OSS-Fuzz and Google subsequently runs fuzzers for those projects continuously and sends reports back to the maintainers. I selected 32 protocol implementations that were using OSS-Fuzz and gathered their development history as recorded by git. By defining and applying some simple rules, I quantified how much effort had been spent on fuzzing for each project. I correlated the fuzzing development history with reported vulnerabilities for each project using the National Vulnerability Database [80]. In doing so, I found that efforts spent on developing fuzz tests seem to pay off in terms of discovered vulnerabilities. However, this was only true for certain types of vulnerabilities. For example, grey-box fuzzing does not seem to help with the discovery of vulnerabilities stemming from the integration of different components or from incorrect calculations (i.e. integer overflows or off-by-one errors). This was the main contribution of Paper A, along with my method and data, and it shed some light on **Q1**. Based on the nature of the underrepresented vulnerability types, I also suggested future directions that fuzzing research could take, with one direction being black-box fuzzing.

In Paper B, I proposed an enhancement to grammar-based black-box fuzzing. Inspired by many grey-box fuzzers’ ability to adapt their input generation based on how the SUT reacts, I set out to model black-box fuzzing as a reinforcement learning problem. Black-box fuzzing, even when using grammars, can be inefficient in the sense that many of the generated test cases do not trigger any interesting behaviour. By introducing a special type of annotation for context-free grammars, I gave a fuzzer the ability to “prune” its grammar during the campaign. The results showed that the fuzzer could learn to prune uninteresting parts of the grammar, thereby letting it fuzz more intensively the parts that were found to be problematic.

My main contributions with Paper B were the formalization of grammar-based black-box fuzzing and showing its practical applicability when using memory usage as a feedback signal. Formalization of the problem helped me make key insights such as what part of the fuzzing setup correspond to a reinforcement learning agent and what are its actions. My first intuition was that the

fuzzer would be the agent and sending fuzzy input would be its actions. But I soon realized that the actions were rather modifying the grammar and the agent was some abstract "external" entity making these modifications. Decoupling the reinforcement learning concept of an action from the act of sending a fuzzy input helped me define a more delay-tolerant reward function. As a short answer to **Q2**, the formal framework of reinforcement learning helped me realize that feedback gathered over multiple test cases was more useful than feedback gathered from a single test case. The feedback consisted of information about process memory usage extracted via non-intrusive system monitoring. It helped the agent discover the most efficient action, i.e. the grammar modification that caused the largest memory consumption in the test subject. While this does answer **Q3**, it only had a noticeable effect in simulated and emulated environments. In more realistic scenarios, the monitoring did not give any noticeable improvement. Nonetheless, with the method described in Paper B, I was able to discover memory leaks in two different implementations of the link-layer discovery protocol [81], one of which was previously unknown.

In Paper C and D I turned my focus towards distributed systems. In Paper A, I had reasoned that grey-box fuzzers struggle to find vulnerabilities arising from improper integration between different software components. The intuition behind this being that they are designed to trace the execution of a single process written in a single language, which is usually not the case for distributed systems. As an example of a distributed system, I chose a generic service-provider network where a number of network nodes are being monitored by a web-based network management system. I found that in such systems, corrupted data can enter the system at unexpected locations via low-level networking protocols and ultimately expose themselves as code injection vulnerabilities in the network management system, thereby answering **Q4**. This was the main contribution in Paper C and I elaborated upon it in Paper D by showing how such vulnerabilities could be exploited to steal credentials or scan internal networks. Lastly in Paper D, I contribute to answering **Q5** by proposing a method for finding these so-called cross-channel scripting vulnerabilities and found several of them in two widely used management systems, one of which was assigned a CVE.

6 Conclusion

If there is one thing that the reader should take away from this thesis, it is that software testing requires creativity. Computer networks and their constituent software systems are ubiquitous in today's society. Yet, there is no uniform way in which they are all developed and deployed. Two expensive enterprise systems could be integrated with a few lines of haphazardly written code. A service that serves millions of users in 2026 could depend on a deployment script that has not been updated since the early 2000's. Every system, be it an individual piece of software or a service-provider network, has a history and requires its own kind of treatment in terms of testing.

In this thesis, I have tried to address this fact with a black-box approach. Any software system needs continuous testing, especially security testing, regardless of its history. Long or short. Straight or crooked. It does not matter, the work of testing is never finished. I chose a black-box approach to fuzz testing because it works with any system. By definition, it does not care about details such as a system's implementation history. Does this mean a black-box fuzzer is a one-size-fits-all solution to security testing? Certainly not! But as a methodology it is always applicable, at least. However, to get the most out of it, the tester needs to be creative. How can the target system be monitored? How can an input format specification be obtained? What are the security implications of unexpected behavior that the target system exhibits during testing? These are all difficult questions that will never get a final answer. Nonetheless, I believe I have come up with some useful intermediate answers in this thesis. I

found that using non-intrusive monitoring of dynamic memory could help improve a grammar-based black box fuzzer. I also found that low-level network protocols could cause injection vulnerabilities in high-level applications. But there is still much work to be done.

In future work I would like to re-think some of the fuzz testing I've done as measurements that can be done on production systems. Fuzz testing should never be done on production systems. This should be self-explanatory; no one wants to risk breaking a deployed network with active users. But having a copy of a large-scale network solely for testing purposes is infeasible in most cases. Some systems have a history and exist under circumstances that simply makes them unique. They still need continuous security testing, though. That is why I would like to stay with a black-box approach while re-designing the generated test cases so that they are harmless. For example, discovering cross-channel scripting vulnerabilities in a service-provider network could be done by deploying a node acts as a part of the network, except that it adds special injection payloads to the traffic it is transmitting. The payloads should be designed in such a way that, if the injection is successful, it causes a browser to send a HTTP request to a dedicated measurement server along with some helpful metadata. To find out if there are any injection vulnerabilities in their production system, the operator can simply check the logs in the measurement server. Or the measurement server can notify the operator via some suitable mechanism. This eliminates the expense and risk of crawling web interfaces in production systems. Moreover, it makes no assumptions about *where* an injection vulnerability might exist, i.e. what system to crawl. Since most systems can afford one occasional extra request, it should be harmless enough to allow for use in production. And while this thesis focused on the LLDP and SNMP networking protocols, there are many other protocols left to be tested for cross-channel scripting vulnerabilities.

Bibliography

- [1] Gunnar Karlsson and Leon Fernandez. “Measuring the Impact of Fuzzing Activity in Networking Software - Extended”. In: *KTH Royal Institute of Technology Technical Reports*. TRITA-EECS-RP. 2025. URL: <https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-372022>.
- [2] Leon Fernandez and Gunnar Karlsson. “Squashing Resource Exhaustion Bugs with Black-Box Fuzzing and Reinforcement Learning”. In: *2023 7th International Conference on System Reliability and Safety (ICSRs)*. 2023, pp. 439–448. DOI: 10.1109/ICSRs59833.2023.10381445.
- [3] Leon Fernandez and Gunnar Karlsson. “Black-box Fuzzing for Security in Managed Networks: An Outline”. In: *IEEE Networking Letters* (2023), pp. 1–1. DOI: 10.1109/LNET.2023.3286443.
- [4] Leon Fernandez and Gunnar Karlsson. “Fuzz Testing for Code Injection Vulnerabilities in Network Management Systems”. In: *2024 8th International Conference on System Reliability and Safety (ICSRs)*. 2024, pp. 529–536. DOI: 10.1109/ICSRs63046.2024.10927541.
- [5] Leon Fernandez and Gunnar Karlsson. “Measuring the Impact of Fuzzing Activity in Networking Software”. In: *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1746–1751. ISBN: 9798400706295. URL: <https://doi.org/10.1145/3672608.3707730>.

BIBLIOGRAPHY

- [6] *Information and Communications Technology Supply Chain Security*. Cybersecurity and Infrastructure Security Agency. Apr. 2025. URL: <https://www.cisa.gov/topics/information-communications-technology-supply-chain-security>.
- [7] Mike Cohn. *Succeeding with agile : software development using Scrum*. eng. The Addison-Wesley signature series. Upper Saddle River, N.J: Addison-Wesley, 2010. ISBN: 0-321-57936-4.
- [8] Daniel Stenberg. *CVE-2020-19909 is everything that is wrong with CVEs*. Aug. 2023. URL: <https://daniel.haxx.se/blog/2023/08/26/cve-2020-19909-is-everything-that-is-wrong-with-cves/>.
- [9] *CVE-2020-21469 is not a security vulnerability*. The PostgreSQL Global Development Group. Aug. 2023. URL: <https://www.postgresql.org/about/news/cve-2020-21469-is-not-a-security-vulnerability-2701/>.
- [10] *testssl.sh*. Dr. Wetter IT-Consulting. Aug. 2025. URL: <https://github.com/testssl/testssl.sh>.
- [11] *The Zonemaster Project*. AFNIC and The Swedish Internet Foundation. Aug. 2025. URL: <https://github.com/zonemaster/zonemaster>.
- [12] *Zed Attack Proxy (ZAP)*. Checkmarx. Aug. 2025. URL: <https://www.zaproxy.org>.
- [13] *Nessus Vulnerability Scanner*. Tenable. Aug. 2025. URL: <https://www.tenable.com/products/nessus>.
- [14] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, et al. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.
- [15] Barton P. Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: <https://doi.org/10.1145/96267.96279>.
- [16] Patrice Godefroid. “Random testing for security: blackbox vs. whitebox fuzzing”. In: *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. RT '07. Atlanta, Georgia: Association for Computing Machinery, 2007, p. 1. ISBN: 9781595938817. DOI: 10.1145/1292414.1292416. URL: <https://doi.org/10.1145/1292414.1292416>.

- [17] Patrice Godefroid, Michael Y. Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. In: *Proceedings of the Network and Distributed System Security Symposium*. San Diego, California, 2008. URL: <https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/>.
- [18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. San Diego, CA: USENIX Association, Dec. 2008. URL: <https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems>.
- [19] Nick Stephens, John Grosen, Christopher Salls, et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”. In: *Network and Distributed System Security Symposium*. 2016. DOI: 10.14722/ndss.2016.23368. URL: <https://doi.org/10.14722/ndss.2016.23368>.
- [20] Sanjay Rawat, Vivek Jain, Ashish Kumar, et al. “VUzzer: Application-aware Evolutionary Fuzzing”. English. In: *2017 Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2017, pp. 1–14. ISBN: 1891562460. DOI: 10.14722/ndss.2017.23404.
- [21] Peng Chen and Hao Chen. “Angora: Efficient Fuzzing by Principled Search”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.
- [22] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, et al. “Scheduling black-box mutational fuzzing”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 511–522. ISBN: 9781450324779. DOI: 10.1145/2508859.2516736. URL: <https://doi-org.focus.lib.kth.se/10.1145/2508859.2516736>.
- [23] Rijnard van Tonder, John Kotheimer, and Claire le Goues. “Semantic Crash Bucketing”. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018, pp. 612–622. DOI: 10.1145/3238147.3238200.
- [24] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, et al. “Optimizing Seed Selection for Fuzzing”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 861–875. ISBN: 978-1-931971-15-7. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert>.

BIBLIOGRAPHY

- [25] Adrian Herrera, Hendra Gunadi, Shane Magrath, et al. “Seed selection for successful fuzzing”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 230–243. ISBN: 9781450384599. DOI: 10.1145/3460319.3464795. URL: <https://doi.org/10.1145/3460319.3464795>.
- [26] Roberto Natella and Van-Thuan Pham. “ProFuzzBench: a benchmark for stateful protocol fuzzing”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 662–665. ISBN: 9781450384599. DOI: 10.1145/3460319.3469077. URL: <https://doi.org/10.1145/3460319.3469077>.
- [27] Haeun Lee, Soomin Kim, and Sang Kil Cha. “Fuzzle: Making a Puzzle for Fuzzers”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3556908. URL: <https://doi.org/10.1145/3551349.3556908>.
- [28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. “Magma: A Ground-Truth Fuzzing Benchmark”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (Nov. 2020). DOI: 10.1145/3428334. URL: <https://doi-org.focus.lib.kth.se/10.1145/3428334>.
- [29] Jonathan Metzman, László Szekeres, Laurent Simon, et al. “FuzzBench: an open fuzzer benchmarking platform and service”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1393–1403. ISBN: 9781450385626. DOI: 10.1145/3468264.3473932. URL: <https://doi-org.focus.lib.kth.se/10.1145/3468264.3473932>.
- [30] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, et al. “Dissecting American Fuzzy Lop: A FuzzBench Evaluation”. In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (Mar. 2023). ISSN: 1049-331X. DOI: 10.1145/3580596. URL: <https://doi.org/10.1145/3580596>.
- [31] Marcel Böhme, László Szekeres, and Jonathan Metzman. “On the reliability of coverage-based fuzzer benchmarking”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1621–1633. ISBN: 9781450392211. DOI: 10.1145/3510003.3510230. URL: <https://doi-org.focus.lib.kth.se/10.1145/3510003.3510230>.

- [32] Chenyang Lyu, Shouling Ji, Chao Zhang, et al. “MOPT: Optimized Mutation Scheduling for Fuzzers”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. ISBN: 978-1-939133-06-9. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/lyu>.
- [33] Stephan Plöger, Mischa Meier, and Matthew Smith. “A Usability Evaluation of AFL and libFuzzer with CS Students”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3581178. URL: <https://doi-org.focus.lib.kth.se/10.1145/3544548.3581178>.
- [34] *Document management — Portable document format*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2020.
- [35] Rauli Kaksonen, Marko Laakso, and Ari Takanen. “Software Security Assessment through Specification Mutations and Fault Injection”. In: *Communications and Multimedia Security Issues of the New Century: IFIP TC6 / TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01) May 21-22, 2001, Darmstadt, Germany*. Ed. by Ralf Steinmetz, Jana Dittman, and Martin Steinebach. Boston, MA: Springer US, 2001, pp. 173–183. ISBN: 978-0-387-35413-2. DOI: 10.1007/978-0-387-35413-2_16. URL: https://doi.org/10.1007/978-0-387-35413-2_16.
- [36] Dave Aitel. *The Advantages of Block-Based Protocol Analysis for Security Testing*. Feb. 2002. URL: https://web.archive.org/web/20221005220401/https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.pdf.
- [37] Pedram Amini, Adam Greene, and Michael Sutton. *Fuzzing: Brute Force Vulnerability Discovery*. eng. Addison-Wesley Professional, 2007. ISBN: 0321446119.
- [38] Joshua Pereyda. *Fuzzing for Humans: Real Fuzzing in the Real World*. DEF CON 24. Aug. 2016. URL: <https://www.youtube.com/watch?v=N3Z4C2D15JM>.
- [39] Michael Eddington. *Peach Introduction*. GitLab. Mar. 2021. URL: <https://peachtech.gitlab.io/peach-fuzzer-community/Introduction.html>.
- [40] Dave Crocker and Paul Overell. *Augmented BNF for Syntax Specifications: ABNF*. RFC 5234. Jan. 2008. DOI: 10.17487/RFC5234. URL: <https://www.rfc-editor.org/info/rfc5234>.

BIBLIOGRAPHY

- [41] Andreas Zeller, Rahul Gopinath, Marcel Böhme, et al. *The Fuzzing Book*. Retrieved 2024-07-01 16:50:18+02:00. CISPA Helmholtz Center for Information Security, 2024. URL: <https://www.fuzzingbook.org/>.
- [42] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. “Grammari-nator: a grammar-based open source fuzzer”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 45–48. ISBN: 9781450360531. DOI: 10.1145/3278186.3278193. URL: <https://doi.org/10.1145/3278186.3278193>.
- [43] Terence Parr. *ANTLR on Github*. ANTLR. Aug. 2025. URL: <https://github.com/antlr/grammars-v4>.
- [44] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, et al. “NAUTILUS: Fishing for Deep Bugs with Grammars”. In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Internet Society, 2019. ISBN: 1-891562-55-X.
- [45] *Protocol Buffers - Google’s data interchange format*. Google. Aug. 2025. URL: <https://github.com/protocolbuffers/protobuf>.
- [46] *OpenAPI Specification*. Swagger. Oct. 2024. URL: <https://swagger.io/specification/>.
- [47] V. Atlidakis, P. Godefroid, and M. Polishchuk. “RESTler: Stateful REST API Fuzzing”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083. URL: <https://doi.org/10.1109/ICSE.2019.00083>.
- [48] Andrea Arcuri. “EvoMaster: Evolutionary Multi-context Automated System Test Generation”. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 394–397. DOI: 10.1109/ICST.2018.00046.
- [49] Myles Foley and Sergio Maffei. “Haxss: Hierarchical Reinforcement Learning for XSS Payload Generation”. In: *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2022, pp. 147–158. DOI: 10.1109/TrustCom56396.2022.00031.
- [50] Michał Zalewski. *Binary fuzzing strategies: what works, what doesn’t*. Google. Aug. 2014. URL: <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>.

- [51] Chad Brubaker, Suman Jana, Baishakhi Ray, et al. “Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations”. In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 114–129. DOI: 10.1109/SP.2014.15.
- [52] Prashast Srivastava and Mathias Payer. “Gramatron: effective grammar-aware fuzzing”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 244–256. ISBN: 9781450384599. DOI: 10.1145/3460319.3464814. URL: <https://doi.org/10.1145/3460319.3464814>.
- [53] Michał Zalewski. *American Fuzzy Lop*. Google. Nov. 2017. URL: <https://lcamtuf.coredump.cx/afl/>.
- [54] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, et al. “AFL++: Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: <https://www.usenix.org/conference/woot20/presentation/fioraldi>.
- [55] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-Based Greybox Fuzzing as Markov Chain”. In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506. DOI: 10.1109/TSE.2017.2785841.
- [56] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, et al. “Directed Greybox Fuzzing”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. URL: <https://doi.org/10.1145/3133956.3134020>.
- [57] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, et al. “Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 337–350. ISBN: 9781450384544. DOI: 10.1145/3460120.3484543. URL: <https://doi.org/10.1145/3460120.3484543>.
- [58] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, et al. “Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols”. In: *Security and Privacy in Communication Networks*. Ed. by Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran. Cham: Springer International Publishing, 2015, pp. 330–347. ISBN: 978-3-319-28865-9.

BIBLIOGRAPHY

- [59] Tammo Krueger, Hugo Gascon, Nicole Krämer, et al. “Learning stateful models for network honeypots”. In: *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*. AISEC '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 37–48. ISBN: 9781450316644. DOI: 10.1145/2381896.2381904. URL: <https://doi.org/10.1145/2381896.2381904>.
- [60] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, et al. “Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner”. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 523–538. ISBN: 978-931971-95-9. URL: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe>.
- [61] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. “AFLNET: A Greybox Fuzzer for Network Protocols”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 460–465. DOI: 10.1109/ICST46399.2020.00062.
- [62] Roberto Natella. “StateAFL: Greybox fuzzing for stateful network servers”. In: *Empirical Software Engineering* 27 (2021), pp. 1–31. URL: <https://doi.org/10.1007/s10664-022-10233-3>.
- [63] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, et al. “Stateful Greybox Fuzzing”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3255–3272. ISBN: 978-1-939133-31-1. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/ba>.
- [64] Shisong Qin, Fan Hu, Zheyu Ma, et al. “NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing”. In: *ACM Trans. Softw. Eng. Methodol.* 32.6 (Sept. 2023). ISSN: 1049-331X. DOI: 10.1145/3580598. URL: <https://doi.org/10.1145/3580598>.
- [65] Eric Olsson, Benjamin Eriksson, Adam Doupé, et al. “SpiderScents: Grey-box Database-aware Web Scanning for Stored XSS”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 6741–6758. ISBN: 978-1-939133-44-1. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/olsson>.

- [66] Nils Bars, Moritz Schloegel, Nico Schiller, et al. “No Peer, no Cry: Network Application Fuzzing via Fault Injection”. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 750–764. ISBN: 9798400706363. DOI: 10.1145/3658644.3690274. URL: <https://doi.org/10.1145/3658644.3690274>.
- [67] *libFuzzer*. LLVM Compiler Infrastructure. Aug. 2025. URL: <https://www.llvm.org/docs/LibFuzzer.html>.
- [68] M.E. Garbelini, C. Wang, S. Chattopadhyay, et al. “Sweyn-Tooth: Unleashing Mayhem over Bluetooth Low Energy”. In: *2020 USENIX Annual Technical Conference*. USENIX, 2020. URL: <https://www.usenix.org/conference/atc20/presentation/garbelini>.
- [69] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, et al. “Automated testing for SQL injection vulnerabilities: an input mutation approach”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 259–269. ISBN: 9781450326452. DOI: 10.1145/2610384.2610403. URL: <https://doi.org/10.1145/2610384.2610403>.
- [70] Erik Trickel, Fabio Pagani, Chang Zhu, et al. “Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2658–2675. DOI: 10.1109/SP46215.2023.10179317.
- [71] Emre Güler, Sergej Schumilo, Moritz Schloegel, et al. “Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4765–4782. ISBN: 978-1-939133-44-1. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/g%7B%5C%22u%7D1er>.
- [72] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, et al. “KameleonFuzz: evolutionary fuzzing for black-box XSS detection”. In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY '14. San Antonio, Texas, USA: Association for Computing Machinery, 2014, pp. 37–48. ISBN: 9781450322782. DOI: 10.1145/2557547.2557550. URL: <https://doi.org/10.1145/2557547.2557550>.
- [73] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. “Black Widow: Blackbox Data-driven Web Scanning”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1125–1142. DOI: 10.1109/SP40001.2021.00022.

BIBLIOGRAPHY

- [74] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, et al. “SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2155–2168. ISBN: 9781450349468. DOI: 10.1145/3133956.3134073. URL: <https://doi.org/10.1145/3133956.3134073>.
- [75] Caroline Lemieux, Rohan Padhye, Koushik Sen, et al. “PerfFuzz: automatically generating pathological inputs”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 254–265. ISBN: 9781450356992. DOI: 10.1145/3213846.3213874. URL: <https://doi.org/10.1145/3213846.3213874>.
- [76] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. “Checking Security Properties of Cloud Service REST APIs”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 387–397. DOI: 10.1109/ICST46399.2020.00046.
- [77] Jonas Bushart and Christian Rossow. “ResolFuzz: Differential Fuzzing of DNS Resolvers”. In: *Computer Security – ESORICS 2023*. Ed. by Gene Tsudik, Mauro Conti, Kaitai Liang, et al. Cham: Springer Nature Switzerland, 2024, pp. 62–80. ISBN: 978-3-031-51476-0.
- [78] Qifan Zhang, Xuesong Bai, Xiang Li, et al. “ResolverFuzz: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4729–4746. ISBN: 978-1-939133-44-1. URL: <https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-qifan>.
- [79] *OSS-Fuzz on Github*. Google. Aug. 2025. URL: <https://google.github.io/oss-fuzz/>.
- [80] *NVD - Home*. National Institute of Standards and Technology. Dec. 2025. URL: <https://nvd.nist.gov/>.
- [81] “IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery”. In: *IEEE Std 802.1AB-2016 (Revision of IEEE Std 802.1AB-2009)* (2016), pp. 1–146. DOI: 10.1109/IEEESTD.2016.7433915.