OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Ville-Veikko Kovalainen

Comparative analysis of fuzzing frameworks and techniques in terms of effectiveness and deployment

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2014

Kovalainen V-V (2014) Comparative analysis of fuzzing frameworks and techniques in terms of effectiveness and deployment. Department of Electrical and Information Engineering, University of Oulu, Oulu, Finland. Master's thesis, 84 p.

ABSTRACT

Fuzz testing, or fuzzing, is a form of software testing where the implementation under test is exposed to unexpected or semi-valid inputs in the interest of robustness testing. Fuzzing frameworks can be utilized to develop fuzzing test suites. In this thesis, a comparative analysis was performed on different fuzzing frameworks including two freely distributed frameworks and one commercial solution.

Three network protocols and a collection of software implementations utilizing these protocols were chosen as test targets. The protocols were File Transfer Protocol (FTP), Locator/ID Separation Protocol (LISP), and Session Initiation Protocol (SIP). A set of test environments was constructed for all the test targets.

Fuzzing test suites for the chosen protocols were developed with the fuzzing frameworks. The test suites were executed towards the test targets and the results monitored closely. A set of metrics for the tests was defined, including effectiveness in terms of found crashes, code coverage, and run-time performance.

The obtained test results show a clear distinction between the tested frameworks. The commercial solution was proven to offer best coverage, found most crashes, and it was overall the most flexible in terms of development and deployment. The others showed promise as well, but lacked in performance or effectiveness, overall quality, and had some shortcomings in development and deployment.

Keywords: fuzz testing, fuzzing framework, software testing

Kovalainen V-V (2014) Vertaileva analyysi fuzz-sovelluskehyksistä ja -tekniikoista suhteessa tehokkuuteen ja käyttöönottoon. Oulun yliopisto, sähkö- ja tietotekniikan osasto. Diplomityö, 84 s.

TIIVISTELMÄ

Fuzz-testaus, eli fuzzaus, on ohjelmistotestauksen muoto, jossa testattavaa toteutusta jykevyystestataan altistamalla se odottamattomille tai osittain valideille syötteille. Fuzz-sovelluskehyksiä voidaan käyttää fuzz-testisarjojen kehittämiseen. Tässä työssä toteutettiin vertaileva analyysi erilaisille fuzz-sovelluskehyksille mukaanlukien kaksi ilmaista testikehystä ja yksi kaupallinen ratkaisu.

Testikohteiksi valittiin kolme verkkoprotokollaa ja kokoelma niitä hyödyntäviä ohjelmistoja. Valitut protokollat olivat File Transfer Protocol (FTP), Locator/ID Separation Protocol (LISP) ja Session Initiation Protocol (SIP). Testikohteille rakennettiin omat testiympäristönsä.

Fuzz-testikehyksien avulla kehitettiin fuzz-testisarjat valituille protokollille. Testisarjoja suoritettiin testikohteita vastaan ja tuloksia valvottiin tarkasti. Testaukselle määriteltiin joukko metriikoita, mukaanlukien tehokkuus löydettyjen kaatumisien muodossa, koodikattavuus ja ajonaikainen suorituskyky.

Saadut tulokset osoittavat selvän eron testattujen fuzz-sovelluskehysten välillä. Kaupallinen ratkaisu tarjosi parhaan kattavuuden, löysi eniten kaatumisia ja oli kaiken kaikkiaan joustavin kehityksen ja käyttöönoton suhteen. Muut vaikuttivat myös lupaavilta, mutta jäivät jälkeen suorituskyvyssä tai tehokkuudessa, yleisessä laadussa ja osoittivat myös puutteita kehityksen ja käyttöönoton suhteen.

Avainsanat: fuzz-testaus, fuzz-sovelluskehys, ohjelmistotestaus

TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

ABBREVIATIONS

1. INTRODUCTION 10

2. SOFTWARE TESTING 11

2.1. Testing purposes 11
2.2. White-box and Black-box 12
2.3. Attack vectors and surfaces 13
2.4. Common bug categories 13
2.5. Mini-simulation 15
2.6. Fuzzing 15
2.6.1. History 16
2.6.2. Why fuzz 17
2.6.3. When to fuzz 17
2.6.4. What to fuzz 18
2.6.5. Fuzzing methods 18
2.6.6. Anomaly types 20
2.6.7. Anti-fuzzing 21

3. FRAMEWORKS AND PROTOCOLS 22

3.1. Fuzzing frameworks 22
3.1.1. Test case generation 23
3.1.2. Injection 24
3.1.3. Instrumentation 24
3.1.4. Reporting 26
3.1.5. Documentation 26
3.2. Peach Community Edition 27
3.2.1. Templating 28
3.2.2. Test case generation 29
3.2.3. Injectors 29
3.2.4. Instrumentation 30
3.2.5. Reporting 32
3.3. Sulley 32
3.3.1. Templating 32
3.3.2. Test case generation 34
3.3.3. Injectors 34
3.3.4. Instrumentation 35

3.3.5. Reporting 35
3.4. Netzob 35
3.5. Traffic Capture Fuzzer 36
3.5.1. Templating 36
3.5.2. TCF SDK 37
3.5.3. Test case generation 37
3.5.4. Injectors 39
3.5.5. Instrumentation 39
3.5.6. Reporting 40
3.6. Tested protocols 40
3.6.1. File Transfer Protocol 41
3.6.2. Session Initiation Protocol 42
3.6.3. Locator/ID Separation Protocol 43

4. TEST SETUP 48
4.1. Research problem 48
4.2. Solution 48
4.3. Metrics 48
4.3.1. Code coverage 49
4.3.2. Normalized Damerau-Levenshtein distance 50
4.4. Testing environment 50
4.5. Test targets 50
4.5.1. File Transfer Protocol servers 51
4.5.2. Locator/ID Separation Protocol targets 52
4.5.3. Session Initiation Protocol User Agent Server 52

5. FUZZER DEVELOPMENT AND DEPLOYMENT 55
5.1. File Transfer Protocol fuzzing test suites 55
5.1.1. Peach 55
5.1.2. Sulley 56
5.1.3. Traffic Capture Fuzzer 56
5.2. Session Initiation Protocol fuzzing test suites 56
5.2.1. Peach 56
5.2.2. Sulley 57
5.2.3. Traffic Capture Fuzzer 58
5.3. Locator/ID Separation Protocol fuzzing test suites 58
5.3.1. Peach 59
5.3.2. Sulley 59
5.3.3. Traffic Capture Fuzzer 60

6. RESULTS 61
6.1. File Transfer Protocol 61
6.1.1. Performance 62
6.1.2. Code coverage 63
6.2. Session Initiation Protocol 64
6.2.1. Damerau-Levenshtein distance 66
6.3. Locator/ID Separation Protocol 67

6.3.1. LISP Data plane 67
6.3.2. LISP Control plane 69
6.3.3. Coverage 69
6.4. Remediation and reporting 70
6.5. Development and deployment complexity 71

7. DISCUSSION 73
7.1. Assessing the results 73
7.2. Known limitations 74
7.3. Future development 74

8. CONCLUSION 76

9. REFERENCES 77

FOREWORD

This thesis work was conducted while working for Codenomicon Oy. The original idea for the research came from the general interested in fuzzing and fuzzing frameworks. The topic found its final form during the summer of 2013, and the research work started during the autumn. The implementation phase was finished during the winter for the most parts, leaving the writing and few final tweaks for the spring. Finally, in May 2014 the thesis was completed.

I would like to thank Professor Juha Röning from Oulu University for supervising the thesis. A huge thanks goes to Codenomicon for coming up with the topic and giving me the chance to perform the research and write the thesis, and especially to Heikki Kortti for supporting and eventually proofreading the thesis.

Finally, I would like to thank my family for believing in me in all the times I did not, giving me the strength to finish what has been the single greatest undertaking of my life so far.

Oulu, Finland May 23, 2014

Ville-Veikko Kovalainen

ABBREVIATIONS

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CLI	Command-Line Interface
CPU	Central Processing Unit
CSV	Comma-separated values
CVSS	Common Vulnerability Scoring System
DDoS	Distributed Denial of Service
DoD	United States Department of Defense
DoS	Denial of Service
DRDoS	Distributed Reflective Denial of Service
EID	Endpoint Identifier
ETR	Egress tunnel router
FTP	File Transfer Protocol
GNU	GNU's Not Unix!
GUI	Graphical User Interface
IH	Inner IP Header
IP	Internet Protocol
ITR	Ingress tunnel router
IUT	Implementation Under Test
LISP	Locator/ID Separation Protocol
LISP-MN	Locator/ID Separation Protocol Mobile node
LoC	Lines of Code
MITM	Man in the middle
OH	Outer IP header
OSS	Open-Source Software
OUSPG	Oulu University Secure Programming Group
PCAP	Packet capture
PMIPv6	Proxy Mobile IPv6
ProFTPD	Pro FTP Daemon
QA	Quality assurance
RAM	Random Access Memory
RDoS	Reflective Denial of Service
RLOC	Routing Locator
RTP	Real-time Transport Protocol
SCTP	Stream Control Transmission Protocol
SDK	Software Development Kit
SDLC	Software Development Life cycle
SDP	Session Description Protocol
SFTP	SSH File Transfer Protocol
SIP	Session Initiation Protocol
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SSH	Secure Shell
SSL	Secure Sockets Layer
TCF	Traffic Capture Fuzzer

TCP	Transmission Control Protocol
TLS	Transport Layer Security
UAC	User agent client
UAS	User agent server
UDP	User Datagram Protocol
USB	Universal Serial Bus
VoIP	Voice over Internet Protocol
WCF	Windows Communication Foundation
XSRF	Cross-Site Request Forgery
XSS	Cross-Site Scripting
xTR	Ingress/egress tunnel router

C	Code coverage
D	Damerau-Levenshtein distance
D_n	Normalized Damerau-Levenshtein distance
L_e	Executed lines of code
L_t	Total lines of code
S_l	String length

1. INTRODUCTION

In the world of today, software is involved in almost everything we do, from making a call using a mobile phone to driving a car with an on-board computer and even controlling power plants or traffic lights. Some software undoubtedly contains security vulnerabilities, since it is written by humans and humans make mistakes. The vulnerabilities might expose the software to attacks. This raises the question whether all of the important software in use today is secure and properly tested.

With that in mind, software testing is becoming more important each day. With the ever increasing number of connected devices, the number of available attack vectors and vulnerable targets is also getting higher than it has ever been before. The purpose of modern software testing is not only to validate the features and performance of the implementation under test (IUT), but also find the bugs and vulnerabilities it contains.[1 p. 3-4].

Fuzz testing, or fuzzing, is a black-box software testing method used to test the robustness of the IUT. As a form of black-box testing, fuzzing tests the functional side of the IUT and does not require any details of the internal implementation of the target [2 p. 21, 83-84]. Fuzzing is performed by sending semi-valid or unexpected inputs to the IUT through its external interfaces [2 p. 24-25]. The purpose of this negative testing is to expose implementation flaws resulting in crashes and other security flaws [2 p. 24-26]. Due to the nature of fuzzing, it is often the best way of finding previously unknown vulnerabilities [2 p. 10-12], such as the bug dubbed as Heartbleed [3] that was recently found in the widely used open source software OpenSSL [4].

There are many tools available for performing fuzz testing, both free and commercial ones. These tools, known as fuzzers, are often specifically designed for fuzzing a single network protocol or a file format. In addition to these stand-alone fuzzers, fuzzing frameworks that can be used to create new custom fuzzers have been developed [2 p. 31]. The frameworks provide a way to define a model or a template for the protocol to be tested, an engine generating the anomalous inputs, and an injector for delivering the inputs to the IUT. The functionality differs greatly between frameworks, many even use completely different fuzzing methods [2 p. 26-29].

Fuzzers, like other testing tools, can be compared in several ways. As a popular software testing metric, code coverage can be used to determine the portion of the source code that is executed during a test run [2 p. 89]. The effectiveness of the fuzzers can be assessed by looking at the found crashes, and performance by measuring the test generation and execution speed.

This thesis presents a comparative analysis of various fuzzing frameworks. The frameworks are compared by first using them to develop fuzzing test suites for three different network protocols, and executing the test suites against a collection of test targets. The main focus for the comparison is on the effectiveness and deployment of the frameworks and the developed test suites. The ultimate goal is to have a clear understanding of the capabilities of each framework, how they compare with each other, and which one of them is the most effective.

2. SOFTWARE TESTING

Software testing can be described as a process of making sure that the tested software product reaches its intended feature coverage, is secure, behaves correctly in all situations, and does not do anything that it was not designed to do. The main goal of software testing must be *bug prevention*. Any variation between the specification and the implementation is a bug, whether it is a simple flaw in the desired functionality or serious vulnerability resulting in a Denial of Service (DoS) state or even permitting an attacker access to the product [2 p. 19]. As complete bug prevention is often impossible due to software complexity and human error, the secondary goal of testing is to *discover* bugs. [5, 1 p. 3-4]

2.1. Testing purposes

As there are many purposes for software testing, there must exist different ways of testing as well. *Conformance testing*, or feature testing, is done to validate whether the tested product conforms with the specifications and implements all of the required features. The main tool for designing the tests is the specification itself, and if it does not implicitly define all of the features, it leaves room for interpretation for both the original developer and the tester, which may lead to misunderstanding and false results[1 p. 34-35].

Conformance testing can be done, for example, by going through the specifications given for the development process and checking if the final product has all of the requested features and that it conforms with all of the specifications. Conformance testing can also include *interoperability testing*, where the tested product is tested with other similar, already existing products by executing them against each other to ensure they can interoperate. [6, 2 p. 17-18, 87]

Performance testing is done to ensure that the product has no performance issues, and that it can function well enough under load. Performance testing may include load and stress testing to see how the product behaves under a high load, for example by using a load generator to simulate multiple concurrent connections or high bandwidth utilization. [7, 8, 2 p. 18, 87]

In order to validate the behavior of a product more effectively, conformance testing is not enough, as it tests only valid or *positive* aspects. The *negative* inputs must also be tested, by what is referred to as *robustness testing*, or negative testing. Robustness testing is done by feeding the product invalid and semi-valid inputs, and monitoring the health of the target. This is done to assess the reliability and robustness of the product under malicious inputs. This is also the most interesting and important part of testing concerning the overall security of the product. [9, 10, 2 p. 18-19, 89-90]

Unlike in conformance and performance testing scenarios, in robustness testing, as the tests contain invalid inputs, the immediate responses from the target are not of primary interest [2 p. 94]. The important question is whether the target can stay functional under invalid, even hostile, inputs. Fuzzing, as described in more detail in section 2.6, can be classified as robustness testing.

After these tests have been done and the product is deemed complete, it is released. However, this does not mean the end of testing. The product will eventually have to be updated, either due to new feature requests or to fix a previously unknown bugs that were not found during pre-release testing. Once the updates are finished, the product must go through post-release *regression testing* to make sure the new fixes or updates did not cause problems in the previous functionality [11]. It should also be remembered, that if the regression tests are not constantly updated while the product itself is updated, they will eventually become ineffective against any new bugs. [2 p. 95-96]

2.2. White-box and Black-box

The box approach is a common way of describing the point of view taken in software testing. *White-box* refers to testing techniques that are testing the structure and internal parts of the software, often called *structural testing*. While doing white-box testing, the test designer has access to the source code, which also provides understanding of the internal structures of the tested software. This can help with designing test cases that target specific parts of the code, and also help make the test coverage as wide as possible. Various testing methods can make use of the source code, both during programming and also while executing the code. For example, white-box testing can be performed by executing the program symbolically, monitoring the executed code paths, and using this information to dynamically generate more test cases that can reach new code paths, theoretically leading to full path coverage. During the testing the program can be monitored using a run-time checker, for example Valgrind [12], for various properties, such as memory errors or threading bugs [13]. [14, 15, 16, 2 p. 80-81]

Other common white-box testing methods may also involve code auditing where the source code is statically checked, either manually by a seasoned developer or by means of automatic source code scanners [2 p. 21-22].

The opposite of white-box testing is black-box testing. *Black-box testing* methods are testing the functionality of the target. Black-box testing does not require access to the source code of the test target, nor any prior knowledge of the internal functionality of the target. The test target is treated as a “black-box”, inside which the tester can not see. Testing is done through the exposed external interfaces, for example input devices and network interfaces. The available interfaces are always included in the possible attack vectors, more about this in section 2.3. [14, 16, 2 p. 80-81, 87-89]

Fuzzing, as will be discussed later, is a particularly powerful form of black-box testing. Vulnerability scanners, like Nessus [17], are also very popular due to their ease-of-use. Vulnerability scanners have tests for finding commonly known vulnerabilities, but nothing besides that, which makes their usefulness limited. [2 p. 36-38].

Testing approach can not always be categorized in such a black and white manner as the techniques mentioned above would suggest. Many times the actual testing approach is a combination between white-box and black-box testing, and the result is often called *gray-box testing*. This refers to a situation where some prior knowledge of the internal functionality of the target is applied into designing the tests in order to achieve the best possible coverage and effect. Gray-box techniques are also often utilized as instrumentation, for example by using a run-time debugger. The test run itself is done as in black-box testing, only through the external interfaces. [14, 2 p. 80]

2.3. Attack vectors and surfaces

Most implementations, whether they are software or hardware, require some form of *input* while they are in use. The input may be, for example, in form of user interaction either via a graphical user interface (GUI) or a command-line interface (CLI), files that the target processes, remote or local network connections, or exposed physical interfaces such as USB or serial ports. In the case of hardware, there is also the possibility of direct hardware tampering. [18, 2 p. 6-9]

All of the input mechanisms, no matter if they are local or remote, are *attack vectors*. Inputs, where privilege changes are happening, are the more interesting ones. As an example, incoming network connections from remote sources may be unprivileged, but in order to process the incoming data, it must be parsed locally and the parser must have some privileges on the host [2 p. 7]. If the parser is vulnerable, and proper counter-measures are not in place, an attacker could achieve privilege escalation [19, 20]. One thing to note on attack vectors concerning network interfaces and other similar cases, such as data input ports, is that a single network interface may in reality be a passageway for multiple different network protocols, each one exposing another service on the target and thus dramatically increasing the number of actual attack vectors originating from a single physical entry point. [18, 2 p. 6-9]

The sum of all attack vectors is called the *attack surface* [18, 2 p. 7]. It covers all parts of the code that a potential attacker can reach, thus making it important from the security testing perspective to cover the whole attack surface when testing an implementation.

2.4. Common bug categories

Knowing what types of bugs exist, and which ones are the most common, helps a tester or a test developer design their tests and anomalies to be more efficient. The same goes for the original developers as well: if they know what kinds of bugs the attackers are going to try and exploit, they can take action during development to mitigate these threats.

Since every implementation is unique, it contains unique bugs that are not seen in any other implementations. However, found bugs and vulnerabilities very often fall into one or more known categories of common vulnerabilities and attacks and their effects, as seen in the list below [2 p. 42-54].

- Memory corruption: Buffer overflows (stack and heap), integer errors, off-by-one errors.
- Web-specific: Injection attacks, Cross-Site Request Forgery (XSRF), Cross-Site Scripting (XSS).
- Timing: Race conditions.
- Denial of Service (DoS) attacks: Including standard DoS, Distributed DoS (DDoS), Reflective DoS (RDoS), and Distributed Reflective DoS (DRDoS).

- Other: Session hijacking, man in the middle (MITM), and cryptography (authentication bypass) attacks.

From these categories, *memory corruption* related problems are possibly the easiest to test for using common testing methods. These are most often found with field level anomalies, as described in section 2.6.6. Especially off-by-one errors [2 p. 47], for example where data written to a buffer is one byte too long, and initialization problems [1 p. 38], where a variable has not been properly initialized, are commonplace since they can easily be left in the code by programmer error. Memory related errors can also be serious threats, since they can often give the attacker a chance for rogue *code execution*, for example through a buffer overflow problem where they can inject their own code to memory. [2 p. 42-49]

As *web applications* have become more popular than ever, they also represent a huge target for attackers, and they also have some specific attacks and possible vulnerabilities that regular applications do not usually have. Injection attacks, including SQL injections and cross-site scripting attacks, are in a sense similar to some regular buffer overflow attacks where the attacker tries to inject and execute their own code. The attack vector in almost half of the web application attacks is a web form [2 p. 8] through which the attacker can bypass any existing input validation and sanitation and inject their code. [2 p. 50-52]

Timing-related bugs, such as *race conditions*, can lead to different types of problems. For example, if a program running on a high privilege is going to read a file from the file system and it first checks if it has the permission to do so, an attacker may overwrite the pointer in the file system to point to another file containing confidential information and thus read the file if the attack is timed correctly [2 p. 53]. Timing-related race conditions may also lead to lockups, if, for example, two separate threads try to access the same resource simultaneously and proper logic for solving race conditions is not in place.

Denial of Service attacks (DoS) can be made in various ways, and protecting against some techniques can be difficult if not impossible. The main goal behind such an attack is to get the target into a *DoS state*, where it can not continue normal service and stops responding to valid queries. This can be achieved in several different ways. The usual approach is to flood the target with valid or invalid requests that allocate so much resources, either local physical resources or network bandwidth, on the target that it simply can not respond anymore. The attack can be performed from a single system, multiple distributed systems (DDoS), or even using a reflective attack where the attacker uses one or more third-party systems to amplify their attack [21].

The categories discussed above cover common bugs, but even those deal only with a portion of the possibilities. Other data processing bugs, architectural, interface and integration problems, operating system errors, and many more still exist, and the first step in mitigating their effects is to know about them. [1 p. 34-54]

One thing to note about different types of bugs and where they reside in the source code is that some bugs can *not* be triggered through the attack surface [22]. These bugs are not usually exploitable and, because of that fact, not as critical as others. Furthermore, these bugs can not be found by fuzzing, which can only cover the external attack surface, but are usually found using other means, such as static code analysis by either automated code analysis tools or a more traditional manual code review. [22]

2.5. Mini-simulation

Mini-simulation is a method for functional *modeling* of protocols and protocol message exchanges. It was developed during the PROTOS project [23], see section 2.6.1, during 1999-2001 [24 p. 4]. Originally, the method was developed for functional robustness testing [24 p. 54-55]. The design goal was to devise a method for generating a large number of protocol messages with only one or few anomalized elements in them at once, leaving the rest of the message unchanged and legal per specification. [24 p. 54-57]

The main idea behind mini-simulation is to create a single protocol entity with multiple small models representing different scenarios in the tested protocol, not one complex model covering everything. This makes for easier maintenance and comprehension, as the smaller models will all be derived from the same protocol specification, and all changes to the overall protocol specification will propagate to the models as well. [24 p. 54-55]

2.6. Fuzzing

Fuzzing is a form of *negative testing* where the test target is fed unexpected or semi-valid inputs or message sequences in order to find out if it can continue valid operations in the presence of anomalous inputs [14, 2 p. 1, 24-25]. A simple example using File Transfer Protocol (FTP) log in sequence can be seen in Figure 1. The first image shows a valid input, and the second image shows a fuzzed sequence, where an anomaly has been introduced to the second message.

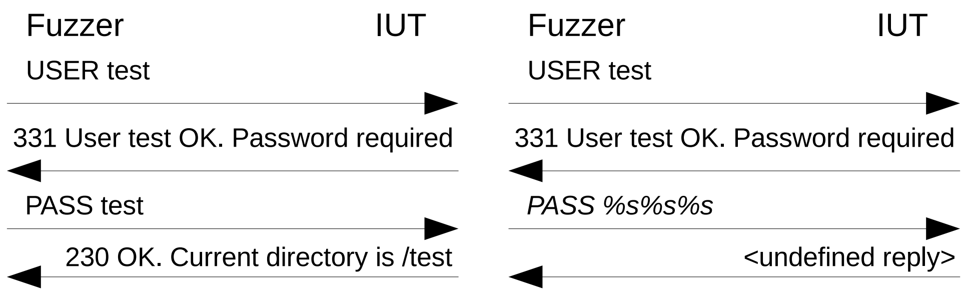

Fuzzer
IUT
USER test
331 User test OK. Password required
PASS test
230 OK. Current directory is /test
Fuzzer
IUT
USER test
331 User test OK. Password required
PASS %s%s%s
<undefined reply>

Figure 1. An example of fuzzer behavior: a) valid input, b) semi-valid input with anomaly in the second request.

Fuzzing is usually performed as black-box or gray-box testing, described in section 2.2, where the tester generally has no access to the internal functionality of the target. The testing can be done through any of the exposed interfaces on the target. This makes it relatively easy to fuzz a target with very little or no knowledge of the internal functionality of the actual target. This also makes the all the findings, by definition, vulnerable to *remote triggering* [22]. [14, 2 p. 1, 22-26, 59]

Fuzzing can be considered *proactive*, as opposed to reactive, security because the main goal is to find unknown vulnerabilities, sometimes referred to as zero-day vulnerabilities, that are not yet used by a potential attackers. Because of this, newly developed protocols and technologies are especially interesting to fuzz since there is no prior knowledge of the possible vulnerabilities, and e.g. known-vulnerability scanners can not be used. [22, 2 p. 10-12]

The usual division of fuzzers is made through fuzzer interfaces, mainly network protocol fuzzing, file format fuzzing, and other, more specialized forms, such as API or serial fuzzing [2 p. 162-165]. For this thesis, the main focus will be on network protocol fuzzing.

2.6.1. History

The early roots of fuzz testing date back to the 1980s. The Monkey [25] was one example of a rudimentary fuzzing-like tool in 1983. The Monkey provided random input through keyboard interaction, effectively acting as a random fuzzer [25, 2 p. 22].

However, fuzzing, as well as other software testing methods, did not gain much popularity until the 1990s when the number of network connected devices and software started increasing, opening up new and easier attack vectors. Before, testing with a security focus was not thought to be so important since without interconnected networks, the attack vectors for possible attackers were scarce. In present days, and especially in the near future, when the number of connected devices is already high and increasing every day, fuzz testing will become a more and more important tool to prevent security incidents by discovering unknown vulnerabilities before the devices and software make their way into production. [2 p. 22-24]

Another factor in the gaining popularity of negative testing and fuzzing was the simplicity of the testing. In 1990, Miller, Fredriksen, and So published a paper on a program called $fuzz$ [26, 2 p. 22] that they had developed, which generates strings of random characters that are automatically fed to the target program through command line input. It was also the first well-known fuzzing software implementation. The $fuzz$ tool, even though it was a simple random fuzzer, found crashes from a wide variety of tested Unix utilities [26], proving the effectiveness of fuzzing.

In 1999, Oulu University Secure Programming Group (OUSPG) launched the PROTOS project [23]. PROTOS project researched different approaches for testing protocol implementations. The testing was done using black-box methods. During the PROTOS project, many different fuzz testing suites were created for a number of network protocols. The best-known was likely the Simple Network Management Protocol (SNMP) test suite [27, 28, 29]. Over 200 vendors were involved with testing their implementation with the SNMP test suite [2 p. 23-26], which was used to discover countless bugs within the SNMP implementations, that were later on patched by the vendors [27, 30, 31, 2 p. 23-26].

Due to the success of PROTOS, the testing community understood the promise in fuzzing. Many new stand-alone fuzzers and fuzzing frameworks were developed, both small one-man projects and extensive commercial solutions. Spike [32] was long the most popular open-source fuzzer, followed by many others such as General Purpose Fuzzer (GPF) [33], and the fuzzing frameworks Peach [34] and Sulley [35]. Commercial vendors took the cue as well: a few of the OUSPG researchers founded Codenomicon [36] and started developing the Defensics [36] fuzzer family, Mu Security (since then changed name to Mu Dynamics and acquired by Spirent [37]) had the appliancebased MU-4000 [2 p. 228], Beyond Security developed beSTORM [38], in addition to some other smaller companies.

2.6.2. Why fuzz

"Why fuzz" is a question often heard in the security world. Many software developers see fuzzing simply as a waste of time and rather spend that time, for example, developing new features. Some developers even to to great lengths to prevent fuzzing on their products; this will be further discussed in section 2.6.7.

These, and other possible reasons aside, fuzzing is often one of the best ways to discover unknown vulnerabilities. Fuzzing works so well because it is great way to cover a wide *input space*, which would often be impossible to achieve manually. The wider the coverage, the more potential vulnerabilities can be found. Fuzzing can also be used against a wide variety of targets and different attack vectors; basically any target that accepts inputs. [2 p. 24, 63]

One of the biggest advantages of fuzzing, as it is a form of black-box or gray-box testing, is that the crashes found through fuzzing are all vulnerable to *remote triggering* and thus they are always real threats. Also, when using a proper instrumentation method, see section 3.1.3, there should not be any false positives; a crash is always a crash. [2 p. 59]

Fuzzing has also proven to be extremely effective compared to manual testing methods because creating the vast amounts of test cases required to reasonably cover the attack surface and input space simply is not feasible. Having an automated system create countless number of test cases to run against a target will eventually find bugs that manual testing would simply have overlooked. [2 p. 63]

Fuzz testing, somewhat depending on the fuzzer of course, is well structured, deterministic, and, most importantly of all, *repeatable*. If a crash found during testing can not be properly detected and repeated, it is not worth much. Some crashes or other bugs, especially those related to high memory consumption, may take more than one test case to trigger them, and reproducing them can be much more difficult. [2 p. 26-27, 30-31]

It must be remembered, though, that fuzzing alone does not solve any problems regarding software stability and security, it is only the first step towards that direction. Fuzzing can be used to discover unknown vulnerabilities, but as such it is only a small component in a complete vulnerability management process. After the flaws have been found, they must be properly reported, fixed, and the fixes deployed into use. [18]

2.6.3. When to fuzz

The short answer to this question is: always. The long answer depends on a few different factors. When developing the software, it is best to integrate fuzzing as part of the used *software development life cycle* (SDLC) in appropriate phases [39, 40]. Usually fuzzing is done in the verification phase (or similar), but it can also be done earlier, for example during the implementation phase. Integrating fuzzing as a mandatory step into the SDLC ensures that fuzzing is done properly in a proper phase of the development life cycle. For example, Microsoft [39, 41, 42] and Cisco [43] have both introduced fuzzing and robustness testing as part of their own SDLCs. [18, 39, 44, 2 p. 11, 25]

One of the main reasons for fuzzing as a part of the SDLC in phases before the actual release is that the *cost* of fixing bugs increases in the later phases of the SDLC, especially after the product is released [45]. The cost of fixing a bug rises exponentially during the SDLC. The cost can rise by a factor of 100 in the release phase compared to the design phase [45, 2 p. 14-15]. If the product is released for end-customers and deployed into production, updating all of the copies will be both difficult and expensive for both the developer and the end-customer. This is an issue especially for specialized embedded devices and their software, which can be hard if not impossible to update after deployment. [18, 44, 45, 46, 2 p. 14-16]

Another reason is the obvious security factor. If the possible vulnerabilities can be found and fixed already during the development process, the attackers can not utilize them when the software is in production and many potential security incidents can be prevented. Also, some bugs that might seem harmless now might be exploitable and cause a severe security breach in the future, when research made by attackers bears fruit and a new method for abusing a previously harmless bug is developed. [44, 47]

However, fuzzing is not only for vendors. The end-customer, or buyer, should also fuzz the products they procure from the supplier or developer [18]. This is done to make sure that the product in question stands up to the standards set by the end-customer. The customer may also request that the products they acquire from a vendor must have been fuzz tested prior to the purchase.

2.6.4. What to fuzz

In short, the answer to this is very similar to the “when to fuzz” question: everything. Special interest should be given to all the products, software or physical devices with inputs that are facing external networks, especially the internet. These products often have the widest and most exposed *attack surface*, see section 2.3.

It is often difficult to say beforehand which products have flaws or will crash when fuzz tested. Common sense says that the more complex a target is, the more likely it is to have one or more flaws. Also, in the field of software security, special attention should be given to programs written in low-level programming languages, especially those with manual memory management [48, 2 p. 42]. Programs written in languages such as C, that require manual memory management are more prone to buffer overflow and underflow attacks, which are exploiting poor memory management. This does not, of course, mean other programs written in high-level languages should be forgotten; almost all software has and will have flaws and they should always be tested. [2 p. 42-54]

2.6.5. Fuzzing methods

There are many different types of fuzzers using alternative methods for fuzzing. Each method has its upsides and downsides. The main categorization of fuzzing methods is made between *mutation* and *generation* fuzzing [2 p. 137]. The most important difference between these is that mutation-based fuzzers have a template of known and valid data for the fuzzed protocol, which they modify based on different techniques. Generation-based fuzzers include a definition of the protocol and, based on the definition, generate the test cases and the test case data by themselves. Both of these main categories can be further split into more specific methods, the most common of which are described below. [48, 2 p. 137-138]

Random fuzzing and random template-based fuzzing are the simplest methods of mutation fuzzing. In *random fuzzing*, the fuzzing engine generates the test cases using random generated data. An example of this is a random bit-flipping fuzzer, that goes through a sample message bit by bit with no regards to keeping the message intact as per protocol specification. That is also the greatest flaw with random fuzzing: most targets will try to validate the incoming message and immediately drop it if the message does not resemble a valid input. Given enough time, random fuzzers will generate messages close enough to a valid input to be accepted by the target, but this process can take a very long time and a vast majority of the generated test cases are more or less pointless. In addition, random fuzzers usually work with a static model or a sample file and do not utilize any dynamic functionality for, for example, copying or calculating critical values during run time, such as lengths and checksums [49], nor do they have any other protocol awareness. [18, 50, 51, 2 p. 26, 138-140]

Template-based fuzzers, also known as block-based or mutational-based, are fuzzers that implement a template for the protocol message or messages. The template is basically a sample message or message sequence of the fuzzed protocol, with elements marking all the different fields and data structures. The generated anomalies are inserted into different elements in the template, keeping the overall structure of the message intact, which also gives template-based fuzzers a clear advantage over pure random fuzzers. Some template-based fuzzers also offer ways to implement some rudimentary dynamic functionality for, for example, calculating checksums [49]. [18, 50, 2 p. 26-27, 138-140]

Template-based fuzzers are limited by the template used to generate anomalies. Only the messages and data structures present in the template can be used when generating anomalized messages. In order to achieve complete protocol coverage, the templates need to cover all possible messages and message sequences. [18, 50, 2 p. 26]

Dynamic generation, or evolution-based, fuzzers use templates similar to template-based fuzzers, but instead of requiring manual input of the templates, dynamic generation fuzzers learn them, either automatically or with little help from a user, by analyzing valid traffic from the target system [2 p. 197-198]. Some can even identify dynamically changing content, for example the mentioned checksums, and calculate values accordingly [49]. This learning process is sometimes called *corpus distillation*. The anomalization itself is done in the same fashion as in template-based fuzzing, where the anomalies are placed in the template keeping the structure intact. [18, 2 p. 26, 138-140]

Model-based, or generational, fuzzers are implemented using a complex protocol model or a simulation, see section 2.5, covering the specification of the tested protocol, or at least the parts required for testing. Model-based fuzzers cover all messages and elements in the protocol, and can handle the dynamic calculations of the stateful features used in the protocol. They simulate the tested protocol by implementing a protocol-aware state-machine, which allows a model-based fuzzer to both handle and fuzz unexpected messages and complex multi-message sequences. This enables a model-based fuzzer to penetrate deeper into the test target and its state-machine, and inject anomalies to parts of code the simpler fuzzers can never reach, thus having a higher chance of discovering vulnerabilities. [18, 50, 2 p. 26-28, 137-140, 142-144]

2.6.6. Anomaly types

As there are different types of bugs, finding them requires multiple different types of anomalies.

The broader look at different anomaly types includes the *anomaly complexity*. The test cases can and should test all of the layers in the implementation under test (IUT), and this means having anomalies that target the different layers in the application logic [52, 2 p. 26]. A list of the most common anomaly complexity categories, and what they are used to test, can be seen below [52, 2 p. 26-27].

- Field level: Used to find field decoding bugs the fields with overflows, underflows, integer or string anomalies
- Structure level: Finding message syntax parsing and validation flaws by changing the structure with overflows, underflows, repetition, and unexpected elements
- Sequence level: Testing the state machine with out-of-sequence and unexpected messages, repetition, as well as omission of messages

Field level anomalies are targeting the parts of the IUT that decode and use the values received in the protocol fields. Depending on the protocol, the values can be in the form of, for example, simple bit values, integers, or complex strings. These can be anomalized with the help an *anomaly library* in the fuzzer, which may contain example anomalies and anomaly types for different field types. For example, integer field anomalies can include common limit values for different integer sizes, off-by-one values, overflows and underflows, or purely random values. [2 p. 26-27, 140-141]

Structure level anomalies reorder the valid structure of the messages, thus targeting the syntax parser and validator of the IUT. The parser tries to crawl through the message assuming it is in a known “language” that is defined in the protocol specification, where all the expected data structures are specified, and tries to interpret the incoming data based on that specification [1 p. 284-288]. If the parser is not robust enough, it can fail while encountering unexpected or missing fields. The structure can be broken in various ways, for example repeating or removing fields, adding unexpected field types, or, if the protocol has fixed field lengths, changing the lengths with overflows and underflows. Some good examples of bugs found by breaking the syntax with structure level anomalies are the ASN.1 related flaws found from SNMP [31, 27] found by the PROTOS project mentioned in section 2.6.1. [2 p. 26-27]

In addition to the field and structure level anomalies that change the message itself, sequence level anomalies are also necessary. They change the sequence of protocol messages, similar to how the structure level anomalies change the structure of one message. Unexpected messages can be added to the message sequence, messages can be repeated, or they can be deleted from the sequence. This aims to confuse the state machine of the IUT, getting it to crash or end up in a deadlock. [2 p. 26-27]

These different types of anomalies can of course be used together with each other to form new combined anomalies. Combining multiple anomalies into one single test case can be effective, but at the same time it makes the identification and remediation of the found problems more problematic. When using a single anomaly it is often fairly straightforward to trace the anomaly into the part or a function in the code where the fault happens. With combined anomalies, the task becomes more difficult, since as there are two or more anomalies, the actual problem in the IUT can lie in multiple different parts of the code, or, of course, a combination of code.

2.6.7. Anti-fuzzing

Anti-fuzzing is an umbrella term for various methods aimed to make fuzzing *less effective*. There have been numerous ways different companies and developers have tried to either make performing fuzz testing against their products too costly or too difficult to achieve. [53, 54]

One of the more famous examples of such a technique is the Anti-Tamper program by the United States Department of Defense (DoD) [53] that is aiming to prevent or delay the exploitation of technologies in the U.S. weapon systems. It is achieving this by trying to deter any reverse-engineering or exploitation against the protected systems. [53]

In the software world, anti-fuzzing has similar goals: to deter the effectiveness of fuzzing, to detect fuzzing, and to prevent the fuzzer from achieving any reasonable test results [54]. These goals are pursued via various means. The developer may build fail-safe systems in their program trying to hide any detectable crashes, or even perform fake crashes and falsifying the instrumentation results if a possible fuzzing attempt is detected [54]. After detecting a fuzzing attempt, the anti-fuzzing methods may also include purposeful performance degradation making the fuzzing process slower and too costly time-wise [54]. All of the methods are aiming to discomfort the fuzzing attempt.

This is also a major problem with anti-fuzzing. As stated, fuzzing is often the best way of finding unknown vulnerabilities and purposefully preventing fuzzing may leave unknown vulnerabilities in place, and thus trusting that the vulnerabilities will be left undiscovered rather than finding and fixing them. If anti-fuzzing is widely deployed, it will result in degradation of quality in software simply because of this.

Anti-fuzzing research is, however, very limited at the moment and there is not enough data existing to properly dictate whether anti-fuzzing is beneficial in the long run [54]. At the same time, there is also research in the opposite direction, making software easier to fuzz. Tavis Ormandy from Google has suggested making software “dumber” so general purpose fuzzers could be used to fuzz and discover bugs from a wider range of software [55]. This would allow more bugs to be found and fixed, and that way increase the overall quality of software [55].

3. FRAMEWORKS AND PROTOCOLS

In this chapter fuzzing frameworks are described. First, an overview is given to what a fuzzing framework is and what it can be used for. Three different frameworks are chosen for closer inspection for this thesis and they are introduced later in this chapter. In addition to the frameworks, the chosen network protocols are also described.

3.1. Fuzzing frameworks

When developing a fuzzer for a custom protocol, the developer most often wants to spend the majority of their time perfecting the protocol *model* or *template*, leaving less focus on the actual fuzzing engine itself. This is why fuzzing frameworks are so popular. A *fuzzing framework* is a collection of utilities that can be used to create a specific protocol model or template and, based on that, generate and deliver anomalized test data to the implementation under test (IUT) [2 p. 31].

There are many fuzzing frameworks available, both free and commercial. The main focus for this thesis will be on the freely distributed frameworks Peach Community Edition [34] and Sulley [35] as well as the commercial Defensics Traffic Capture Fuzzer from Codenomicon [36]. The main differences between the frameworks, apart from the commercial factor, are the programming and markup languages, as well as the protocol definition grammar, used for modeling the target protocol or file type and the fuzzer itself, and of course the used fuzzing method for generating anomalies.

Fuzzing frameworks contain all of the needed components and utilities for making a working fuzzer. A list of the most essential parts of a typical fuzzing framework can be seen in the list below [2 p. 29-30]. Of course, a working fuzzer does not per se need all the parts shown here, and not every framework implements all of them.

- Model: A user-editable protocol model or a template used for defining the message structures and message sequences for the tested protocol.
- Anomaly library: A collection of “effective” inputs to use as anomalies. For example, variable boundary values, known vulnerability triggers, control characters. Anomaly library can also be a simple random generator.
- Test case engine: An engine that creates the actual anomalized test messages using the protocol model as base and adding anomalies from the anomaly library.
- Injector: A method for delivering the test cases to the target.
- Instrumentation: A way to monitor the health of the IUT during test run.
- Reporting: After a test run has been completed, any found flaws should be documented and reported. A tool or feature for generating reports is needed.
- Documentation: Especially in the case of fuzzing frameworks, as opposed to ready-made fuzzers, comprehensive documentation is a must-have feature in order to develop a working fuzzing test suite.

Fuzzing frameworks, with the exception of pure mutation fuzzers that generate random data with no structures, have ways of defining a *model* or *template*, using a framework-specific protocol definition grammar, that represents the data structures that are used in the protocol in question. The frameworks also contain a test case *engine* that creates, possibly with help from a separate anomaly library, the actual test cases containing anomalies based on the protocol model. The test case engine is the most important part of the frameworks. As the input space and amount of possible combination anomalies is practically infinite, the engine must be smart enough to create a reasonable amount of anomalized test cases that still achieve a wide test coverage [18, 56]. This is explained in more detail in section 3.1.1. [2 p. 29-31]

The frameworks also need an *injector* that is used to deliver the anomalized, fuzzed test case data to the target [2 p. 222-223]. Many fuzzing frameworks also provide some sort of utilities and methods for *instrumentation*, that means checking the health of the target during test runs [2 p. 223]. Comprehensive documentation is also an important feature for anyone using a new fuzzing framework to build a test suite, as well as reporting the found problems. These will be further explained in the next sections. [2 p. 29-31]

3.1.1. Test case generation

As previously discussed in section 2.6.5, there are several ways fuzzing can be performed. In the end, again with the exception of pure random fuzzers, they all perform in a similar way: start with a valid message and anomalize a part of it.

The valid message, that is needed as a base, can be either a template or a sample file (template-based fuzzers), or it can be dynamically generated from pre-determined specification (model-based fuzzers). That valid message is then passed on to the fuzzer engine.

The fuzzing engine is the most important part of a fuzzer or a framework. The engine creates the *anomalies* that are used for the actual testing. It reads the valid message, or sequence of messages, and adds the created anomalies, ending up with a semi-valid message sequence [2 p. 222]. That sequence makes up a test case. The fuzzing engine may have an anomaly library containing examples of inputs known to trigger vulnerabilities, common limit values, or random data [2 p. 29]. Experience in the security and testing fields, and a good understanding of common programming paradigms, help with designing an effective anomaly library. Different anomaly types are discussed in section 2.6.6.

It is also important to note that when an anomaly is inserted into a test case, especially if the test case contains a sequence of multiple messages, the test is obviously not valid per specification anymore. This also means that whatever should happen in the test case sequence after the anomaly is not defined in the specification. Due to this, it is not important what the implementation under test replies *after* the anomaly is sent since there is no correct behavior specified. Whether the test case is a pass or a fail is determined later using instrumentation, as discussed in section 3.1.3.

The *quality* of anomalies mainly depends on the engine. The used model or template has an effect in determining how well the anomaly can be placed in the message, that is if the structure of the message is kept intact [2 p. 222]. As mentioned before in section 2.6.5, if the structure is not valid per specification, there is a high chance that the target simply drops it without ever trying to process it, thus rendering the test case pointless.

The engine also has to take into account the number of test cases. The *input space* is practically infinite, which means there are no limits to the amount of possible anomalies, especially with combination anomalies, thus making the amount of possible test cases infinite in return [56]. In order to keep the amount of test cases and the resulting test run time reasonable, the engine must be *intelligent* enough to choose which anomalies are worth trying and which can be discarded [56]. Here again the importance of a properly populated anomaly library is emphasized.

3.1.2. Injection

Once the engine has created an anomalized test case, it has to be delivered to the test target. An *injector* provides a method for sending the test case data. In the case of network protocol fuzzing, an injector can be a piece of code that automatically connects to the target in the beginning of each test case, sends the test case data, and also receives incoming data if the sequence includes incoming messages. Most fuzzing frameworks by default have multiple injectors, for example IPv4, IPv6, TCP, or UDP. [2 p. 222-223]

An injector does not have to be network-based: in case of file fuzzing, the test cases could be simple files saved into the file system. Any type of attack vector can, at least in theory, have its own injector.

3.1.3. Instrumentation

Instrumentation is a term used for the method of checking and monitoring the *health* of the target during a test run. *Instrumentation* is a key factor for getting meaningful test results. As mentioned in section 3.1.1, after an anomaly is sent to the target, the sequence is not valid or within specification anymore, so the responses from the target can not be used to determine a pass or a fail verdict without further instrumentation. Without any kind of instrumentation, it would be impossible to detect failures and crashes and pinpoint them to a certain test case or series of test cases that trigger them, rendering the testing close to useless. [18, 2 p. 223]

As there are many kinds of targets running in various environments, there must be many different ways of doing instrumentation. The most common instrumentation methods are process-based instrumentation, some form of system instrumentation, and valid-case instrumentation. In addition to these common methods, especially in the case of embedded devices or proprietary systems, a custom method of instrumentation developed specifically for the target in question is often needed for obtaining most accurate results.

Process instrumentation

When using *process instrumentation*, the target process (program) is started within a controlled environment where it can be monitored for crashes and incoming signals. The simplest forms of process instrumentation detect if a process crashes or is otherwise shut down during the test run and report this as a failed test case. More advanced methods may include attaching a debugger to the target, which is used to catch exceptions and monitor the target in more detail. [2 p. 176-180]

Most process-based instrumentation methods are also capable of controlling the state of the target process: starting it, restarting it after a crash, returning it to its original known state, and killing it. This can be used when executing long automated test runs.

The major downside with process instrumentation is its platform dependency: if the monitor process can not be run on the target system, it is not possible to use process instrumentation. Because of this, process instrumentation is not a possibility when testing, for example, embedded devices or programs running in other closed, proprietary systems, or when there simply is no access to the target system.

System instrumentation

In system instrumentation, the logs and resources in the system where the target is running monitored. For example, if the target has proper logging features, checking log files for both the target program and the system itself for warning and error messages can reveal crashes or other problems other instrumentation methods can not catch. [2 p. 171-172]

Monitoring the resource consumption can also be used as instrumentation. This can include checking the elapsed CPU time, monitoring reads and writes to the file system, and more [2 p. 171-175].

Similar to process instrumentation, the biggest downsides of system instrumentation are that it is platform dependent, and that its effectiveness depends very highly on the way the target software has been built.

Valid-case instrumentation

Valid-case instrumentation is one way of countering the problem with process and system instrumentation. In valid-case instrumentation, when fuzzing a network protocol, a simple *valid query* (test case) is sent to the target using the same protocol that is under test and the reply is checked and compared to a known valid reply. This will let the fuzzer know if the IUT is still up and able to respond to a valid query. It should be noted that valid-case instrumentation is only useful with protocols that implement at least one request-reply sequence; if the target is never expected to reply anything, valid-case instrumentation is of no use. [18, 2 p. 170-171]

The advantages of this over some other instrumentation methods is that it implicitly checks the health of the target protocol and process, and not just the overall health of the system or the process [18]. The downside, compared to process instrumentation, valid-case instrumentation can miss some crashes, for example in the case if the IUT has a watchdog that watches the state of the service being tested and quickly restarts it after a crash before a valid test case can be sent. [2 p. 170-171]

Other instrumentation methods

The three methods mentioned above are the most commonly used, but not by far the only methods. Simple Network Management Protocol (SNMP) [57] is one good example. SNMP is meant for monitoring remote systems, which makes it a fitting choice for an instrumentation method.

There are some more advanced methods including library interception [2 p. 180-181], binary simulation [2 p. 181-183], source code transformation [2 p. 183], and the use of development tools, including debuggers and memory analyzers.

Especially in the case of highly customized proprietary targets, it is often necessary to come up with a custom instrumentation method. A good example is a networking device, a router or a switch, that can not easily be instrumented using the usual methods. In this case, a custom instrumentation method could be developed which establishes a separate remote connection to the target and checks the health using internal methods, similar to system instrumentation.

All in all, the used instrumentation method must always be one that works with the target in question. As the targets vary, so must the instrumentation methods do as well.

3.1.4. Reporting

Reporting is an important part of any type of testing. Once bugs have been found, they must be *reported* to the responsible party. On a smaller scale, if the developer is the one doing the testing by themselves, the importance of reporting decreases. But in any situation where the testing is performed either by a separate quality assurance (QA) team or a third party tester, the importance of proper reporting increases to a whole another level. In order for the responsible party to identify, reproduce, and eventually fix the bug, the party must be provided with accurate information about the test case that triggered the bug, as well as about the testing environment. [2 p. 223]

The report can be in many different forms, including plain log files, automatically generated reports with graphs and statistics to help the reader better understand the process, and more. Some testing software, for example the commercial beSTORM fuzzer [38], can also export test cases to external scripts or binaries that can be used to reproduce the found bug without the need for the original testing software.

As mentioned in section 2.6.2, fuzzing, as well as other kinds of software testing, is only the first step in a vulnerability management process. Bugs are found by testing, but in order to get them fixed, a good reporting system is a tremendous help. A descriptive report with accurate details is the best way of helping the responsible party, be it an internal developer or a third party vendor, identify and mitigate the found problems. [18]

3.1.5. Documentation

While building and using a fuzzing test suite, the provided documentation is in a big role. This covers not only the traditional usage instructions, but also the test case documentation, which is often neglected at first glance.

When at first developing the test suite, the documentation of a framework is an indispensable resource. A comprehensive API documentation is needed, as well as instructions on how to configure the test run, instrumentation, and everything else related to testing. Having up-to-date documentation should be a given, but with the modern agile SDLCs, where focus is given to functionality over comprehensive documentation, this is unfortunately often not the case, as will be seen in later sections of this thesis.

In addition to the traditional documentation, when using fuzzing test suites a descriptive test case documentation is a commendable feature. The fuzzing engine creates countless anomalous test cases, and, after a bug has been found and the bug is being reported, information about the anomaly is needed. This information should include at least the description of the added anomaly in such a detail that it can be reproduced without the original framework. Some frameworks can, for example, show a packet decomposition where the anomaly is marked.

Test case documentation may also include additional information for helping the tester evaluate the seriousness of the found bug or vulnerability. Some frameworks provide a Common Vulnerability Scoring System (CVSS) score, which is the industry standard for measuring software vulnerabilities, which gives a fair idea of the possible threat level [22, 58].

3.2. Peach Community Edition

Peach Community Edition (further referenced as Peach) is a free and quite popular fuzzing framework developed by Deja vu Security [34]. While not technically an open source project, the source code of Peach Community Edition is freely available under MIT license [59]. Peach performs template-based fuzzing, with some elements of model-based or generation fuzzing included, based on special Peach Pit files that are used to define the template for generating fuzzed test cases. Peach offers a wide range of injectors, including IPv4 and IPv6, TCP and UDP, and more. It also comes bundled with various instrumentation methods, called Monitors. Peach is written in .NET [34, 60] and runs on mono [34, 61], which in turn means that Peach can be run in any operating system where mono is available, for example Windows, Linux, and OS X. [34]

There are three different editions of Peach available: Community, Professional, and Enterprise. Of these three, the Community edition is free and the other two are commercial and require a paid license. The biggest differences come from the subscription to the Peach Pit Library, which contains ready-made Peach protocol models, access to enhanced product documentation, and the ability to use Peach Advanced Fuzzing Engine [62]. This thesis will focus on the Community edition and self-made protocol templates.

Peach offers a few additional tools besides the basic fuzzer. Among these are a file fuzzer FuzzBang with a GUI, a dumb network fuzzer NetworkFuzzer also with a GUI, and a Validator which can be used to validate a Peach Pit file and visualize the data in the template. The Peach fuzzing framework itself does not have an official GUI for creating nor running tests using user-created Pit files.

3.2.1. Templating

In Peach, special Pit files are used to define the template. The Pit files used with Peach are written in XML format, which makes validating the syntax very easy.

Peach's protocol model, which in reality acts more like a template (see 2.6.5), is split into two parts: the data model and the state model. All of the different messages, and parts of the messages, are presented as data models. Data models contain the structured data fields of the message, type of the data (string, integer, etc.), and it may also contain special *hints* that the anomalizer engine can use to make better-suited anomalies for the field in question. An example of a simple data model can be seen in Listing 1.

```
<DataModel name="DM_USER">
  <String name="command" value="USER"/>
  <String name="space" value=" " token="true"/>
  <String name="parameter" value="test"/>
  <String name="endline" value="\r\n" token="true" mutable="false"/>
</DataModel>
```

Listing 1. Peach $DataModel$ example

The sample code in Listing 1 shows how Peach can be used to model a data element, identified as $DataModel$, in this case an FTP USER message. The $DataModel$ is built from various built-in element types, for example the $String$ element. The various $DataModel$ elements make up the messages for the tested protocol. They can be used to represent both outgoing messages that will be anomalized, and incoming messages that can be parsed to match a $DataModel$.

Peach also offers various helper functions, called Fixups and Transformers, for multiple dynamic functionalities that can be embedded in the template. These include, for example, functions for calculating Crc32 checksums, MD5 hashes, compression algorithms like Gzip, and various type transformations and encoding routines. There is also ScriptFixup which allows an external Python script to be used to modify the template on the fly.

The previously created $DataModel$ elements are used when defining the protocol states, as seen in Listing 2.

```
<State name="Quit">
  <Action type="output">
    <DataModel ref="DM_QUIT"/>
  </Action>
    <Action type="input">
      <DataModel ref="DM_FTP_response"/>
    </Action>
    <Action type="close"/>
</State>
```

Listing 2. Peach $State$ example

In Listing 2 is an example of a $State$ element, consisting of multiple $Action$ elements that are used to, for example, send and receive data. $State$ elements are defined as sets of $Action$ elements, which make up the state transformation. In the above example, the $State$ defines a sequence of messages for sending FTP QUIT command and receiving a reply.

A *StateModel* in Peach consists of one or more *State* elements. It is used to represent all of the states and essentially it makes up the test case sequence. A simplified example of a *StateModel* is shown in listing 3.

```
<StateModel name="SM_LOGIN" initialState="Login">
  <State name="Login">
    <Action type="connect"/>
    <Action type="input">
      <DataModel ref="DM_FTP_response"/>
    </Action>
...
    <Action type="changeState" ref="Quit"/>
  </State>
  <State name="Quit">
...
  </State>
</StateModel>
```

Listing 3. Peach *StateModel* example including multiple *State* elements

The *StateModel* first defines the initial state where the test case execution starts. Within the *State* elements, the Action elements are used to move to the next state when needed.

3.2.2. Test case generation

Peach uses a mix between generational (model-based) and mutational (template-based) test case generation. Walking through the template defined in the Pit file, Peach uses special *mutators* to mutate the data and to create anomalies. The mutators make up the fuzzing *engine* of Peach.

Peach has dedicated mutators for different built-in element types, for example *Array*, *String*, and *Numerical*. There are also mutators for producing structural anomalies by duplicating, removing, or re-ordering *DataElements*. A complete list of mutators is available in the Mutator section in Peach's online documentation [63].

In addition to the mutators, Peach also allows configuration for the overall mutation strategy. The strategy effects the ordering of cases, whether it is random or sequential or something in between, and the creation of *combined* mutations.

The mutators and the overall mutation strategy can be configured in the Pit file.

3.2.3. Injectors

Peach offers injectors, that are called Publishers in Peach, for the most common data link, network, and transport layer protocols, as well as file formats and direct console outputs [34]. A complete list can be seen in Table 1.

Table 1. Default injectors in Peach

Injector	Description	Notes
Com	Allows calling methods and properties on COM objects	Windows only
Console	Output data to standard out	
ConsoleHex	Output data to standard out as prettified hex format	
File	Read or write an external file	
FilePerIteration	Creates output file for every iteration (test case)	
Http	Sends data over HTTP	Supports authentication, cookies, and SSL
Null	Discards all data	Used for unit tests
RawEther	Sends raw Ethernet packets	Linux recommended
RawIPv4	Sends raw IPv4 packets with IP header	Linux recommended
RawIPv6	Sends raw IPv6 packets with IP header	Linux recommended
RawV4	Sends raw IPv4 packets without IP header	Linux recommended
RawV6	Sends raw IPv6 packets without IP header	Linux recommended
Remote	Sends using remote Peach Agent	
TcpClient	Connects to remote TCP service	
TcpListener	Listens for incoming TCP connections	
Udp	Sends and receives UDP packets	
WebService	Allows calling SOAP and WCF based web services	

The most notable missing publisher is SCTP [64], used, for example, in Diameter [65]. In addition to default publishers, Peach also supports self-made publishers, written in .NET.

3.2.4. Instrumentation

There are various instrumentation mechanisms, called Monitors, provided with Peach. Most of them are meant directly for target health monitoring, like process and crash monitors, but there are also some convenience Monitors, such as the SSH [66] Downloader that can be used to, for example, fetch remote log files related to a crash. All the available Monitors are listed in Table 2.

Table 2. Default instrumentation methods (Monitors) in Peach

Monitor	Description	Notes
Windows Debugger	Controls debugger instance	Windows only
CleanupRegistry	Removes a registry key	Windows only
PageHeap	Enables page heap debugging options for an executable	Windows only
PopupWatcher	Closes windows based on their title	Windows only
WindowsService	Controls a Windows service	Windows only
CrashWrangler	Controls Crash Wrangler tool, monitors for crashes	OS X only
CrashReporter	Reports crashes detected by OS X System Crash Reporter	OS X only
LinuxCrashMonitor	Registers with kernel and catches faulting processes using GDB [67]	Linux only
CanaKitRelay	Controls a set of relays for turning AC and DC lines on/off	
CleanupFolder	Removes contents from a directory after each test case	
IpPower9258	Controls networked power switch	
Memory	Monitors the memory usage of a process	
Pcap	Takes a network capture during test run	
Ping	Validates if target is up using ping (ICMP Echo)	
Process Launcher	Controls a process during test run	
ProcessKiller	Kills specified processes after each test case	
SaveFile	Saves a specified file after a fault	
Socket listener	Monitors for incoming TCP or UDP connections	
SSH	Connects to remote host over SSH to run commands	
SSH Downloaded	Downloads files over SSH SFTP [66]	
Vmware Control	Controls VMware [68] virtual machines	

The Monitors in Peach are run using an Agent process, that can be used either locally or remotely. An Agent can run one or more Monitors. A remote Agent allows Peach to do, for example, process health monitoring in a remote system, and the results from the Monitor are relayed back to the Peach process doing the actual fuzzing.

Peach also has a feature for automatically trying to reproduce and pin down the exact test case or sequence of cases that caused instrumentation to fail. Peach does this after an instrumentation failure by iterating the test cases backwards and checking if the detected fault can be reproduced. This is done by first trying the previous test case by itself, and if it does not work, trying the two previous test cases one after another, and so on. If this feature is left on, the faults reported at the end of the test run are all reproducible.

An important thing to note about the Monitors in Peach is that all of the Monitors that are checking the target health in order to give a test case verdict, save perhaps the Ping Monitor, are run as processes in the target system. This requirement makes instrumenting embedded or proprietary systems and devices, for example network routers and switches, extremely difficult if not impossible.

3.2.5. Reporting

Peach does not by default offer any reporting functions besides simple logging of test cases into a file. It should be noted that Peach prints the element being fuzzed and the mutator used for the fuzzing to the command line, but it does *not* write this information to the log files, which makes analyzing the results very difficult. It is possible to extend the logging feature by developing a custom Peach extension used for logging.

3.3. Sulley

Similar to Peach, Sulley is a free open source fuzzing framework performing template-based fuzzing based on user-written templates. Sulley is developed by Pedram Amini, Aaron Portnoy, and Ryan Sears. The core of Sulley is written in Python [69], and the protocol templates used with Sulley are also written in Python, which makes it extremely easy to take advantage of scripts and even external programs while modeling as you can use any Python functions in the middle of the model. This is a big advantage compared to most other fuzzing frameworks especially when modeling protocols that have multiple similar message structures. [35, 70, 71]

Sulley is also a multi-platform fuzzing framework, supporting platforms where Python can be run, for example Windows, Linux, and OS X. Sulley shares much of its syntax and ideology with the older, and quite popular at its time, fuzzer SPIKE made by Immunity [32]. The aim of Sulley is to offer powerful data generation, yet at the same time maintain simplified data presentation and transmission [70, 71].

Sulley does not offer any real UI for designing or running the tests other than the Python scripts created manually by the tester. There is, however, a run-time web UI that shows the progress of the tests, and list of found crashes.

3.3.1. Templating

Sulley is similar to Peach in a sense that the data elements, referred to as requests in Sulley, and state/message sequences are defined individually. With Sulley, the template is created using Python. Sulley provides, similarly to the others, various classes, called *primitives*, for templating certain kind of elements, for example s_string for modeling strings as seen in the example request in Listing 4. [70, 71]

```
s_initialize("USER")
s_string("USER")
s_delim(" ")
s_string("test")
s_static("\r\n")
```

Listing 4. Sulley request example

One major problem with Sulley's built-in primitives is that there is no bit-sized primitive at all. The output will always be padded to the next byte. This, with the problem of missing lower layer injectors discussed later, makes it impossible to create a template for a protocol with fields smaller than a single byte.

One of the advantages of Sulley, when it comes to creating the template, is that it also enables the use of regular Python expressions. For example, when creating a template for a repetitive pattern, or multiple similar data structures or requests, a regular Python for loop can be used to simplify the design. An example of creating requests for similar FTP commands using a for loop is shown in Listing 5.

```
noattr = ["CDUP", "QUIT", "REIN", "PASV", "STOU", "ABOR"]
for command in noattr:
    s_initialize(command)
    s_string(command)
    s_static("\r\n")
```

Listing 5. Python loop example for creating Sulley requests

This kind of declaration of the model saves tremendous amounts of time and effort compared to traditional methods, for example in Peach, where every command has to be modeled individually.

One of the other advantages of Sulley is the State Graph definition that Sulley uses for connecting different request and state transformations. With Sulley, the requests are connected to each other as seen in Listing 6.

```
s.connect(s_get("USER"))
s.connect(s_get("USER"), s_get("RealQUIT"))
s.connect(s_get("RealUSER"))
s.connect(s_get("RealUSER"), s_get("PASS"))
s.connect(s_get("PASS"), s_get("RealQUIT"))
s.connect(s_get("RealUSER"), s_get("RealPASS"))
```

Listing 6. Connecting requests to a state model in Sulley

The connected states form a graph of states, or messages, that also defines the message sequence for each test case. During test execution, Sulley will walk through the graph, testing each request at a time. When getting further down in the graph, Sulley will prefix the tested command with the ones before it in the graph. An illustration of the graph is shown in Figure 2. For example, when testing $PASS$ request in the examples, $RealUSER$ always precedes it. [70, 71]

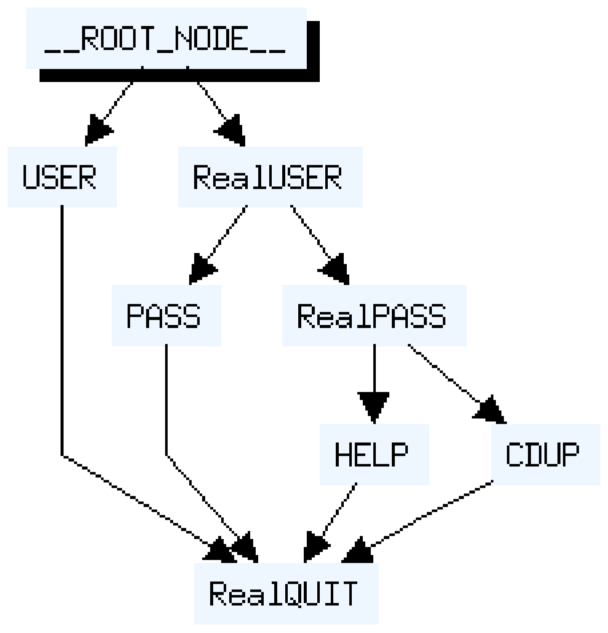


Figure 2. Graph made of request connections in Sulley.

3.3.2. Test case generation

Compared to Peach, Sulley creates the test case sequences in slightly different way. As mentioned, when creating the model or template for a protocol, the requests are interconnected to form a state graph. Sulley iterates through the graph covering and fuzzing all of the different requests in a sequence, resembling transaction flow graph testing [1 p. 121].

3.3.3. Injectors

Sulley has a very limited collection of injectors. A list can be seen in Table 3.

Table 3. Default injectors in Sulley

Injector	Description
TCP	Send or receive TCP packets
UDP	Send or receive UDP packets
SSL	Send or receive SSL [72] encrypted traffic over TCP

What is notable here is that Sulley does not have any data link layer or network layer injectors, which means it impossible to fuzz any protocols on those lower layers. There is at least one third-party modified version of Sulley, called L2Sulley [73] by Enno Rey and Daniel Mende from ERNW [74], that supports injecting to lower network layers.

For this thesis, only the original Sulley is used, and the third-party versions are deemed out of scope.

3.3.4. Instrumentation

For instrumentation, Sulley does not offer many choices by default. Sulley's default instrumentation methods, called Agents, are shown in Table 4.

Table 4. Default Agents in Sulley

Agent	Description
Netmon	Monitors network traffic and saves packet captures (PCAP)
Procmon	Process monitor for checking target health
VMControl	Network API for VMware instrumentation and control

The only instrumentation method capable of giving a pass or a fail verdict in Sulley is Procmon. Similar to Peach, this has to be run as a normal process in the same OS as the target. Again, this poses a problem with instrumenting when testing an embedded device, or a device with a proprietary OS.

3.3.5. Reporting

Similar to Peach, there is no ready-made report generation in Sulley, other than the simple log files. There are some convenience tools for examining the test results in more detail, but no real reporting function is present.

3.4. Netzob

Netzob is a tool meant for reverse engineering, and generating, or simulating, network protocol traffic. Netzob provides the tools for creating a model for a protocol by learning its message format and state machine by means of active and passive traffic analysis. The model can be used in simulating traffic, and it can also be used as a template source in fuzzing. Netzob can export Peach Pit files, which Peach can in turn use in fuzzing as explained in section 3.2, based on the extracted protocol models. [75]

In terms of fuzzing, Netzob is dependent on Peach, and it can be considered more of an extension than competitor to Peach. Netzob's main advantage is the template creation. With Peach alone, the template must be written completely manually. Netzob does this automatically, or with some input from the user, after the protocol message or message sequence has been learned from real traffic. In this sense, Netzob is very similar to Traffic Capture Fuzzer.

3.5. Traffic Capture Fuzzer

Traffic Capture Fuzzer [76], or TCF, is a commercial fuzzing suite distributed and sold by Codenomicon [36]. It uses a combination of model-based, template-based and dynamic fuzzing. TCF differs from most other fuzzer frameworks in the way that its protocol model definition is created.

From an user interface point of view, TCF offers a clear-cut graphical user interface (GUI), a command-line interface (CLI), and a HTTP Application Programming Interface (API), the latter two being helpful for headless systems and automation purposes.

3.5.1. Templating

With TCF, a packet capture (PCAP) is used as a basis for the protocol template. The fuzzer creates a sequence file based on the capture, similar to how Netzob can create a Peach Pit file. The capture is analyzed using Wireshark's dissectors [77]. TCF then creates the message template and message sequence based on the analyzed data. This template can be further edited manually if needed. An example of a TCF $< send >$ block can be seen in Listing 7.

```
<send name="ftp-message2">
  <content>
    <label t="response">
      <label t="request">
        <protocol p="ftp">
          <label t="ftp">
            <label t="ftp_request_command">'USER'
            </label>' '
            <label t="ftp_request_arg">'test'
            </label>'\r\n'
          </label>
        </protocol>
      </label>
   </label></content>
</send>
```

Listing 7. An example of a $< send >$ block in TCF

The block defines the message format and that it is an outgoing message (send). The full sequence can have multiple $< send >$ blocks as well as $< recv >$ blocks for defining incoming message.

As the example shows, TCF does not specify any special fields for different types of inputs. TCF automatically recognizes the field type from the field notation. For example, in the above example, the text "USER" is in single quotes, which equals to string input. TCF can detect other types of inputs as well, for example hex (0x1234) and bits (0b0101).

Using an automatically analyzed packet capture as a starting point allows an extremely quick test suite development time for that particular message sequence. The downside with this approach is, as with other template-based methods, that when testing a complex protocol with multiple different messages and message sequences, the number of needed packet captures for different scenarios increases.

3.5.2. TCF SDK

Traffic Capture Fuzzer with Software Development Kit (TCF SDK) [76, 78] is an addition to the normal TCF. The SDK can be used with TCF to further extend the test suite functionality using Java code. TCF SDK exposes an Application Programming Interface (API) that provides access to the messages sent and received during testing, and allows customization of the messages and logic [76].

An example of how the SDK integrates with the normal Traffic Capture Fuzzer can be seen in Figure 3. Custom Java code can be used to control the outgoing messages using the SDK after the test case engine has created the case, that is after an anomaly is inserted, and incoming messages both before and after the engine has parsed and handled them.

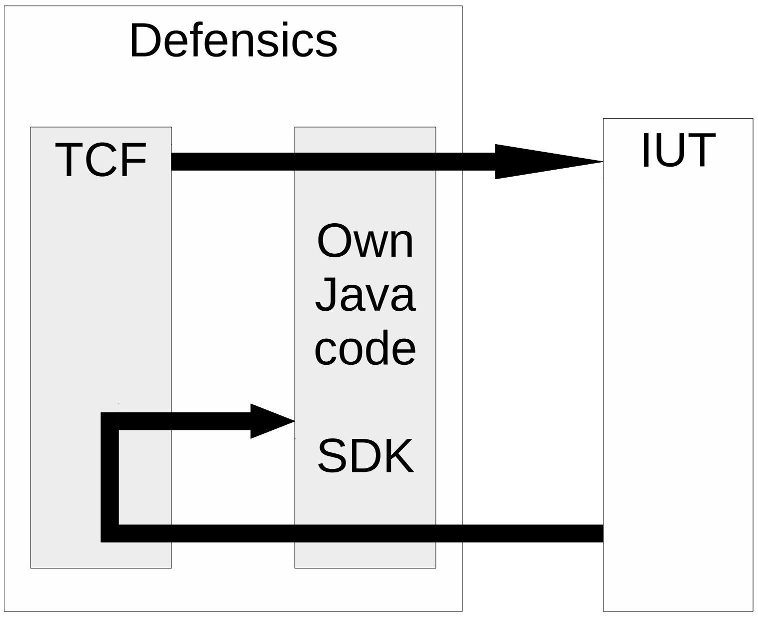


Figure 3. SDK integration with Traffic Capture Fuzzer.

The SDK can be utilized for various purposes. For example, modifying outgoing messages by changing dynamic values, calculating hashes and checksums, etc. The SDK also enables dissecting the structure of the incoming messages manually and extracting field values, as well as performing setup and tear-down for both test runs and single test cases. [76]

With the features of the SDK, it is possible to extend the basic static template-based fuzzing of TCF to include more dynamic functionality and even state-awareness, bringing the end-result closer to model-based fuzzers.

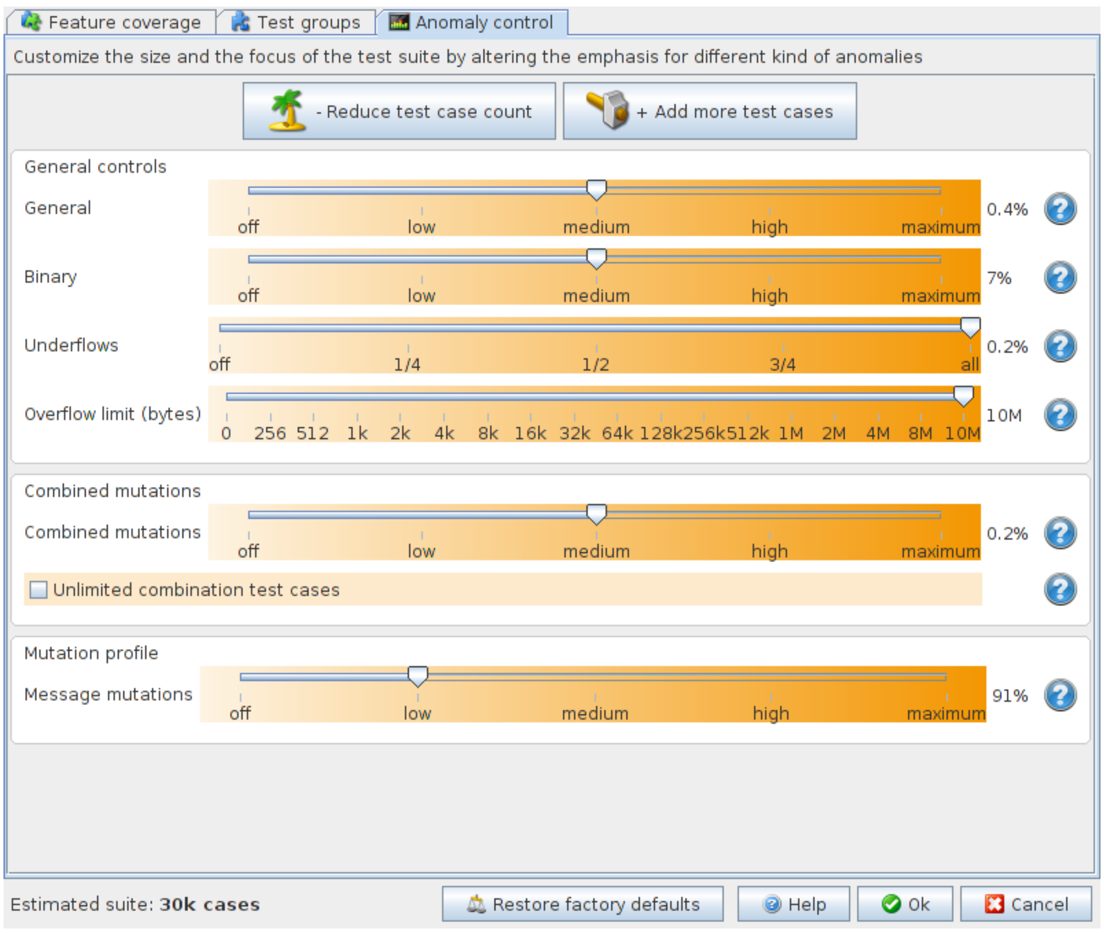
3.5.3. Test case generation

Traffic Capture Fuzzer, as a Codenomicon Defensics [36] test suite, follows the usual model-based generation of other Defensics test suites to a certain degree. As the model, or template, used in TCF is limited to what the original capture included and no deeper protocol knowledge exists, TCF must employ template-based mutational test case generation as well [52].

The fuzzing engine creates anomalies based on the input fields. The anomaly types include field level anomalies, for example overflows and underflows, structure level anomalies, for example element repetition and non-default element inclusion, as well as message sequence anomalies. TCF's engine can also create binary anomalies, such as bit flipping and bit terminations, and textual anomalies, for example format strings, control characters, and other textual anomalies. The engine can also create combination anomalies by adding two or more anomalies in the same test case. The created test cases will be inserted into test groups based on the location of the anomaly within the message.

In addition to these, TCF's engine creates mutational test cases. The mutational cases are created by mutating the imported valid case using a set of mutation fuzzing engines, which use the created template to create test cases intelligently. The mutational test cases are placed in an "Extended" test case group, separate from the normal groups. Also, the test case information, and the test case view, is not as comprehensive for mutational cases as it is for the regular ones.

TCF offers the possibility to control the anomalization, which means that the user can choose which type of anomalies should be used and the amount which they should be generated. An example from GUI view is seen in Figure 4.



Customize the size and the focus of the test suite by altering the emphasis for different kind of anomalies

General controls

General: medium 0.4%

Binary: medium 7%

Underflows: all 0.2%

Overflow limit (bytes): 10M

Combined mutations

Combined mutations: medium 0.2%

☐ Unlimited combination test cases

Mutation profile

Message mutations: low 91%

Estimated suite: **30k cases**

Figure 4. Anomaly control in Traffic Capture Fuzzer.

3.5.4. Injectors

When importing a valid network capture into TCF, the user can choose whether to use raw Frame replay or Application replay mode. If Application replay mode is used, TCF will create the template using the application layer data and use the built-in injectors for handling the transport and network layers below. With Frame replay, the whole frame is templated as it appears in the capture and transmitted raw.

A complete list of injectors is shown in Table 5.

Table 5. Default injectors in Traffic Capture Fuzzer

Injector	Description
Ethernet	Send or receive raw Ethernet frames
IP	Send or receive raw IP packets
TCP	Send or receive TCP packets
UDP	Send or receive UDP packets
SCTP	Send or receive SCTP packets

In addition to the common TCP and UDP, TCF is the only one of the three frameworks that has support for SCTP as a transport layer injector. With the help of the Software Development Kit, the list of injectors can be further extended. For example, handling SSL encryption could be added, similar to what Sulley offers by default.

3.5.5. Instrumentation

TCF offers a few different ways of instrumentation by default. Most notable, it offers Valid-case instrumentation that is not present in the other frameworks. A list of the available instrumentation methods can be seen in Table 6.

Table 6. Default instrumentation methods in Traffic Capture Fuzzer

Method	Description
Valid-case	Confirms correct behavior using a valid-case for the tested protocol
SNMP	Checks SNMP objects and monitor for traps
Connection-based	Checks if a (TCP) connection to the target can be made after a test case
External	Allows for any external scripts or programs to be run as instrumentation

Compared to, for example, Peach, the list is short. TCF is also the only one of the tested frameworks that does not have a process-based instrumentation method by default. During this thesis, a method for remote process-based instrumentation is developed, that can be used as external instrumentation. TCF's external instrumentation can be used to run any scripts or programs as instrumentation or at other parts of the test run, and the return values from the external instrumentation programs are used to determine the pass or fail verdict.

TCF also has the valid-case instrumentation, see section 3.1.3. In valid-case instrumentation, after an anomalous test case is sent, the test suite sends a valid message or message sequence using the same protocol that is under test. This is done to ensure that the target is still capable of normal functionality. For example, when testing FTP, a simple ping check might still get a valid response from the target even if the FTP implementation under test has itself crashed, but if a valid-case instrumentation is used and a valid FTP message is sent, the crash will be noted.

In addition to this, TCF also does connection-based instrumentation, if TCP is used as a transport protocol, automatically after each test case is sent. The instrumentation is done at the same time as the next test case is started, meaning if the TCP connection for the next test case can be established, the connection-based instrumentation for the previous case is considered as passed.

Process-based instrumentation

Both Peach and Sulley have an instrumentation method meant for remote process monitoring. For this thesis, a similar process monitoring script will be written in Python [69] to be used with TCF as an external instrumentation method. The script should allow remote process monitoring for crashes, kill signals, and similar. The script should also allow automated run control of the target, which means it can be used to remotely start or kill the target process. This in combination with the options available in TCF's External instrumentation allows for fully automated test runs where the target process is started automatically at the beginning, and can be restarted after a crash.

3.5.6. Reporting

TCF offers a few different reporting choices. These include simple reports with the test run information and a list of test cases (test case name and group) with verdict, and detailed machine-readable statistics in CSV format.

TCF can also export more detailed, and formatted, PDF or DOC format reports, which include information about the test run, verdict, and detailed information about the failed test cases, as well as instructions on how to interpret the data.

3.6. Tested protocols

The chosen test protocols for this thesis are File Transfer Protocol (FTP) [79], Session Initiation Protocol (SIP) [80], and Locator/ID Separation Protocol (LISP) [81]. These particular protocols were chosen for various reasons. FTP is an older, simple yet very popular protocol for file transfers with a wide variety of possible test targets ranging from the extremely simple and lightweight to more complex and feature-rich implementations.

SIP, the same as FTP, is a popular protocol with many available implementations. SIP was chosen alongside FTP since it is a slightly more complex protocol, and it has some dynamic functionality that will give some challenge to the capabilities of the used fuzzing frameworks.

LISP, on the other hand, is a new lower layer routing protocol. As a new protocol, still being under development, LISP does not offer as wide a range of available test targets as FTP and SIP, but it is still an intriguing protocol to fuzz since it has not yet gone through thorough testing and thus might also attract attention from possible attackers. Creating an accurate template for LISP will also be a more difficult challenge for the frameworks.

3.6.1. File Transfer Protocol

File Transfer Protocol (FTP) is one of the oldest protocols for file transfer still in active use. The original specification, RFC 114 [82], was published in 1971. Back then, FTP was transmitted over Network Control Program (NCP), which preceded TCP/IP which is the norm in today's internet [82].

The original FTP over NCP was replaced by specification RFC 765 using TCP/IP as transport in 1980 [83]. RFC 765 was obsoleted in turn again in 1985 by RFC 959 [79], which is also the version of FTP in use today.

In its current form, FTP is a TCP protocol that is built using a client-server architecture. The control connection and communication in FTP is done in clear text format and using a set of predefined commands. A list of the available commands can be seen in the specification [79] section 5.3.1. An example of FTP log in process over the control connection is shown in Listing 8. In addition to the control connection, FTP uses a separate data connection for the actual file transfers. [79]

```
  220 Welcome to FTP server.
USER test
  331 User test OK. Password required
PASS test
  230-User test has group access to:  test
  230 OK. Current directory is /home/test
```

Listing 8. FTP Log in sequence example consisting of USER and PASS commands

The file transfers in FTP can be done using either *active* or *passive* mode. In active mode, the client listens for an incoming data connection from a server, whereas in passive it is the server that listens for the incoming data connection from the client. The data connection itself can be made as either in stream mode (stream the data through without processing), block mode (data is broken down into blocks), or compressed mode (the data is compressed using some algorithm). [79]

As for access control features, FTP supports both user-based authentication and anonymous access. This poses a potential security problem because, as mentioned earlier, the login data is sent over the control connection in clear text. There are extensions to the original FTP protocol that define some added security. For example, the proposed standard FTPS [84] defines FTP transfers using Transport Layer Security (TLS) [85] to secure the connection between the endpoints.

3.6.2. Session Initiation Protocol

Session Initiation Protocol (SIP) is an application-layer control protocol for signaling communications. It is most often used for setting up and configuring multimedia sessions, for example setting up voice over IP (VoIP) calls. [80]

SIP can be used to create, modify, and terminate sessions. The sessions can have one or more participants. SIP handles the session creating using invitations (INVITE message), which provide the necessary information needed for the other party to both negotiate the used media types and other parameters as well as to join the session. On a more technical side, SIP can run on top of different transport-layer protocols, namely UDP, TCP, and SCTP. For the scope of this thesis, UDP will be used as transport. [80]

The information in SIP, similar to FTP, is transmitted in clear text, with the aim of being human readable. An example of a SIP INVITE message can be seen in Listing 9.

```
INVITE sip:102@172.16.0.101 SIP/2.0
Via: SIP/2.0/UDP 172.16.0.101;branch=z9hG4bK1398074206513;rport
Content-Length: 338
Contact: <sip:1011@172.16.0.101;transport=udp>
Call-ID: 21125702673-9-Case-0@172.16.0.101
Content-Type: application/sdp
CSeq: 9 INVITE
From: "user"<sip:1011@172.16.0.101>;tag=1398074206496-Test-0
Max-Forwards: 70
To: <sip:102@172.16.0.101>
User-Agent: SIP Test Suite
```

Listing 9. Example of a SIP INVITE message

In SIP communications, there are two main user agents present: user agent client (UAC) and user agent server (UAS), which form a typical user-server architecture. In most real-world applications, for example softphones, the protocol implementation handles the functionality for both an UAC and an UAS, which means it is able to both initiate and receive the first connection.

As SIP is only used for signaling and session setup, it is often used in conjunction with another protocol which is used for the actual data, such as a video stream. The media information is usually carried using Session Description Protocol (SDP) [86], whereas Real-time Transport Protocol (RTP) [87] is often the choice for the video or audio itself. The SDP part is added to the SIP message as payload. An example of an SDP payload defining various supported audio modes is shown in Listing 10.

```
v=0
o=- 3367046830 3367046830 IN IP4 172.16.0.101
s=SIPSuite
c=IN IP4 172.16.0.101
t=0 0
a=direction:active
m=audio 49152 RTP/AVP 3 97 98 8 0 101
a=rtpmap:3 GSM/8000
a=rtpmap:97 iLBC/8000
a=rtpmap:98 iLBC/8000
a=fmtp:98 mode=20
a=rtpmap:8 PCMA/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-11,16
```

Listing 10. Example of an SDP payload for SIP message

Most SIP implementations, especially softphones, require media information to be present in an incoming SIP INVITE message, or it will be rejected.

3.6.3. Locator/ID Separation Protocol

Locator/ID Separation Protocol (LISP) is a relatively new routing protocol aimed to help with some of the shortcomings observed in the currently used routing protocols. At the time of writing this thesis, LISP specifications are still officially considered as *draft* or *experimental*. [81, 88, 89]

The main idea behind LISP's separation of Locator and ID data is that currently the internet architecture combines two functions, routing locators, telling where you are in the network, and identifiers, telling who you are, into one number space that is the IP address. By *separating* these two functions, the aim of LISP is to improve *scalability* of the routing system. The internet routing system was and is not scaling well enough when faced with the recent growth rate of new web sites and the ever increasing number of connected devices. [81, 88, 89, 90]

Decoupling the *routing locator* (RLOC) and *endpoint identifier* (EID) specifiers is a logical choice, as the data stored in them simply can not be used efficiently if it is combined into a single identifier. In order to maximize the routing efficiency and improve aggregation of the RLOC space, the routing locators should be assigned based on network topology. On the other hand, in order to keep the nodes effectively manageable and retain persistent identity for end-nodes, the identifier should not be tied to topology and to changes in topology. This is especially relevant with mobile nodes that can often travel from one network to another. In this sense, LISP is somewhat similar to Proxy Mobile IPv6 (PMIP or MPIv6), which is a network-based mobility management protocol that allows mobile nodes to have IP address and session continuity when roaming from network to network [91]. Another example in which the decoupling of locator and identifier data would be helpful is with the more and more popular multi-homed sites. The routing and addressing systems currently in use simply do not work well and scale with multi-homed sites that are not part of any single network topology. [81, 88, 89, 90]

LISP achieves this functionality by *encapsulating* the regular IP packets as LISP packets at the edge of a network, which are then routed based on LISP's mapping databases. The original host first sends a packet as it would normally do. An ingress tunnel router (ITR) at the edge of the network where the packet originated encapsulates the packet as a LISP packet, and sends it to the egress tunnel router (ETR) at the edge of the destination network. The ETR strips the LISP headers and the packet is then routed as usual inside the destination network to the target host. More details on the encapsulation will be shown in section 3.6.3. An important thing to note for real-life situations is that the ITR and ETR can both reside on the same router, which is then called an xTR. [81, 89]

One of the biggest advantages of LISP is its easy deployment: LISP can be *incrementally deployed*. LISP does not require a change to core internet structure or all hosts; the changes can be incrementally deployed one host at a time. As noted above, LISP is also designed to be deployed only to the edge routers in a network, leaving other hosts completely unchanged. When sending out network packets, LISP enabled routers check their assigned mapping databases if the target host matches a known RLOC. These databases can be updated using map requests to mapping servers. If a known RLOC includes the target EID (host), the router adds a LISP header to the packet and delivers it to the target RLOC address. If the original target host can not be found in the mapping database, the packet is simply routed as any other non-LISP package. [81, 88, 89]

From here two distinct functionalities can be seen: routing the encapsulated packets and gathering control information (for example, map requests). LISP splits these into *data plane* and *control plane*, respectively, somewhat similarly to FTP. For the purposes of this thesis, the focus will be on the LISP encapsulation targeting the data plane, and encapsulated map requests targeting the control plane.

Thus far, from the major router manufacturers, Cisco has been the one most interested in deploying LISP [92, 93, 94, 95].

LISP encapsulation

When a router (ITR) encapsulates an outgoing packet to a LISP packet, it adds the required headers in place and transmits the package. This takes place on the data plane. On a technical level, the header of a LISP packet is constructed with an outer IP header (OH), a UDP header, a LISP header, and the inner IP header (IH). An important thing to note is that LISP supports the EID and RLOC being different IP versions, both IPv4 and IPv6. At the time of writing this thesis, LISP specification has support for all four combinations of IPv4 and IPv6 as outer or inner protocol, that is a LISP packet can have IPv4-in-IPv4, IPv4-in-IPv6, IPv6-in-IPv4, and IPv6-in-IPv6. Figure 5 [89] shows an example of IPv4-in-IPv4 data plane packet structure. [89]

```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     / |Version|  IHL  |Type of Service|          Total Length         |
    /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   |         Identification        |Flags|      Fragment Offset    |
   |   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   OH  |  Time to Live | Protocol = 17 |         Header Checksum       |
   |   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   |                    Source Routing Locator                     |
    \  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     \ |                 Destination Routing Locator                   |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     / |       Source Port = xxxx      |       Dest Port = 4341        |
   UDP +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     \ |           UDP Length          |        UDP Checksum           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   L   |N|L|E|V|I|flags|            Nonce/Map-Version                  |
   I \ +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   S / |                 Instance ID/Locator-Status-Bits               |
   P   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     / |Version|  IHL  |Type of Service|          Total Length         |
    /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   |         Identification        |Flags|      Fragment Offset    |
   |   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   IH  |  Time to Live |    Protocol   |         Header Checksum       |
   |   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   |                           Source EID                          |
    \  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     \ |                         Destination EID                       |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

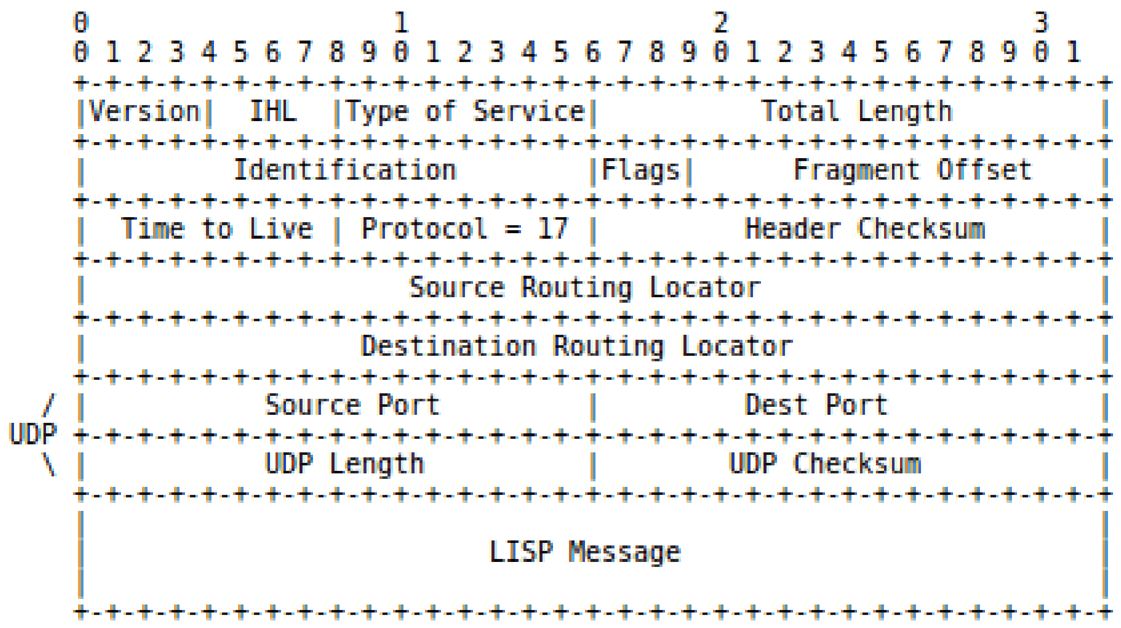
Figure 5. IPv4-in-IPv4 encapsulated LISP data plane packet format. IHL represents inner header length.

When first sending the packet, the ITR encapsulates the packet with the needed headers. The OH will have the RLOC addresses as source and destination, so the packet will be routed to the correct ETR. That is followed by a standard UDP header, which is followed by the LISP header. The LISP header includes various LISP-specific flags (nonce present, locator-status-bits field enabled, echo-nonce-request, map-version, instance id bit, and a 3-bit field for future flag usage), a nonce value, instance id value, and the locator-status-bits. [89]

After the LISP header comes the inner IP header that uses the original (EID) values as source and destination. And lastly after the header, the actual encapsulated message payload is sent. When the packet reaches the ETR at the other end, it is decapsulated by stripping all of the excess headers, and routed as a normal IP packet inside the target network. [89]

LISP map request

LISP map requests are encapsulated similarly to data plane packets, as shown in the previous section. The map request itself is a LISP control plane packet consisting of an IP header, UDP header, and the LISP payload. An example of an IPv4 control plane packet can be seen in Figure 6 [89].



```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |Version|  IHL  |Type of Service|          Total Length         |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |         Identification        |Flags|      Fragment Offset    |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |  Time to Live | Protocol = 17 |         Header Checksum       |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                    Source Routing Locator                     |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                 Destination Routing Locator                   |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     / |           Source Port         |         Dest Port             |
   UDP +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     \ |           UDP Length          |        UDP Checksum           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                                                               |
       |                       LISP Message                            |
       |                                                               |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6. LISP control plane packet structure.

An ITR uses the LISP map request for querying a mapping for an EID, testing if the RLOC can be reached, or to refresh cached mapping information [89]. LISP map request is transmitted as a payload in the encapsulated control plane packet as seen in Figure 6. The structure of a LISP map request can be seen in Figure 7 [89].

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |Type=1 |A|M|P|S|p|s|    Reserved     |   IRC   | Record Count  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                         Nonce . . .                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                         . . . Nonce                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |         Source-EID-AFI        |   Source EID Address  ...     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |         ITR-RLOC-AFI 1        |    ITR-RLOC Address 1  ...    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                              ...                              |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |         ITR-RLOC-AFI n        |    ITR-RLOC Address n  ...    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  / |   Reserved    | EID mask-len  |        EID-Prefix-AFI         |
Rec +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  \ |                       EID-Prefix  ...                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                   Map-Reply Record  ...                       |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7. LISP map request packet structure.

A map request, in addition to the mandatory flags and control information, contains information about the source EID, ITR RLOC which is the ITR that actually sends the request, as well as the queried EID. A more detailed explanation of the packet structure is available in the LISP specification [81].

It should be noted that when performing a cache refresh, or checking if an RLOC can be reached, the source EID can be omitted [89].

4. TEST SETUP

In this chapter, the test environment is defined. This includes defining the research problem, solution, used metrics, and detailed information about the test targets.

4.1. Research problem

The main purpose of this thesis is to perform a comparative analysis of various fuzzing frameworks. This is done with effectiveness and deployment as the main focus points. The goal is to have a clear understanding of what each framework is capable of, how do they compare with each other, and which produces the most effective tests.

4.2. Solution

In order to produce reasonable test data for determining the quality of the tested fuzzing frameworks, a collection of *test suites*, fuzzers for a certain protocol, are developed using various frameworks for different network protocols. The chosen protocols are discussed in detail in section 3.6. The test suites are executed against a set of targets while the targets are instrumented for crashes. The targets are defined in section 4.5.

The results are measured using a set of metrics, which are described in more detail in section 4.3. The main focus will be on effectiveness, in terms of amount and quality of anomalies as well as instrumentation, ease of use and deployment.

4.3. Metrics

For comparable results, a set of metrics is needed. Most often the metrics used within the realm of software testing are used to measure which testing technique is the most effective, or if the test suite working hard or smart, meaning if the ratio of test cases to number of bugs found is smaller or higher [1 p. 215-216]. In addition to these basic concepts covering effectiveness, this thesis concentrates also on the development and deployment factors, in terms of complexity and the more subjective metric of ease-of-use. The items in the list below are considered while assessing the results.

- Development complexity
- Effectiveness: Found flaws (crashes), test case generation
- Normalized Damerau-Levenshtein distance
- Coverage
- Deployment
- Remediation

The used metrics for measuring the effectiveness of the developed fuzzing test suites will cover the amount of crashes in the test targets, the effectiveness of the available instrumentation methods, meaning if they catch the failures and if they can be utilized in all of the scenarios, the overall quantity and quality of the test material using Damerau-Levenshtein distance as a measure, as well as the performance when injecting the test cases to the target.

For the performance tests, only a single fuzzing thread will be run simultaneously. Parallel or distributed test execution would provide better performance. The enterprise version of Peach does have support for distributed runs and Sulley lists parallel runs as an upcoming feature, but for the current versions there is no support.

4.3.1. Code coverage

Code coverage, including line, branch, and function coverage, is one of the more popular and used methods of measuring the utilization of the target program in both real life usage scenarios as well as testing purposes. Code coverage measures how much of the original source code is actually executed when the compiled program is run. When put into use in fuzz testing scenarios, code coverage gives fairly good statistics of how deep into the target program's inner functionality the fuzzer can penetrate and how large a portion of the program is actually executed. [14, 96, 97, 98, 2 p. 89, 130-133, 225-226]

Code coverage using lines of code (LoC) can be calculated as a ratio of executed LoC to total LoC, as can be seen in Equation 1.

$$C = \frac{L_e}{L_t} \tag{1}$$

In the equation, L_e represents the amount of executed LoC, L_t the total LoC, and C is the line coverage. Branch coverage and function coverage are calculated in the same manner, comparing the number of executed branches or functions to all of the ones existing in the source code.

There are, however, some cases when code coverage as a metric for black-box or gray-box testing might give misleading results. Software often has sections of code that simply can not be reached through the attack interface. In these cases, code coverage will give accurate results of the actual executable code covered, but this result can be far off if compared to attack surface coverage, see section 2.3. [2 p. 130-133, 198-199, 244-246]

There are also many different tools for measuring code coverage. The tools are often language-specific, meaning that a particular tool can be only used to measure coverage for a program written in a certain programming language. For C, one often used is the free and open-source gcov [99], from the GNU Compiler Collection (GCC) [99]. In addition to gcov, there are also the commercial C Test Coverage Tool by Semantic Designs [100] and BullseyeCoverage Code Coverage Analyzer [101].

For this thesis, *gcov*, and its front-end lcov [102], will be used as the code coverage measurement tool.

4.3.2. Normalized Damerau-Levenshtein distance

The Damerau-Levenshtein distance is a metric for measuring the *edit distance* between two sequences or strings. It is a combination of Levenshtein distance [103], which is calculating insertion, substitution and deletion operations, and Damerau distance [104], which includes the operation of transposing two adjacent characters. That is, the Damerau-Levenshtein distance between two strings is calculated as the minimum number of single character insertions, substitutions, deletions, and transposing two adjacent characters, needed to transform one string to another [103, 104, 105].

For this thesis, a normalized Damerau-Levenshtein distance is used. The normalized value D_n is calculated as the ratio of the Damerau-Levenshtein distance D to the length of the longer compared string S_l. A ratio of 0 denotes that the compared strings are identical, whereas two strings with nothing in common will get the value 1. This is shown in Equation 2.

$$D_n = \frac{D}{S_l} \tag{2}$$

As mentioned before in section 2.6.5, the anomalized messages generated by the fuzzing engines should resemble valid messages, since completely random messages would more than likely be discarded by any IUT, and they should not be too far of off that mark. The Damerau-Levenshtein distance is used to measure how different the anomalized messages are compared to the valid messages.

4.4. Testing environment

The tests will be run in an isolated and virtualized environment. The used virtualization environment for this thesis is Oracle VM VirtualBox 4.3 [106]. The test targets are run in Ubuntu 13.04 Linux distributions [107] virtual machines (VM), with the exception of the Session Initiation Protocol (SIP) target SJphone which is run in 64-bit Windows 7. Mode detailed test target configurations are shown in section 4.5 below. Code coverage and other process-based instrumentation methods are also run on the same VM as the targets.

The fuzzers themselves are run natively in Ubuntu 13.10. The virtual network connecting the fuzzers to the target VMs is isolated from other network interfaces for the duration of testing. All the tests are run on a high end personal laptop computer (Intel i7-3520M CPU, 8GB RAM).

4.5. Test targets

When choosing the test targets, preference was given to open-source software (OSS) that could be compiled in Linux, especially when choosing FTP and LISP targets. This was due to the chosen code coverage measurement methods, namely gcov, mentioned in section 4.3.1.

4.5.1. File Transfer Protocol servers

File Transfer Protocol, being an old and extremely popular network protocol, has a wide range of implementations available. For these tests, the chosen targets will be Pure-FTPd [108] and Pro FTP Daemon [109]. Both are widely used and are considered to be relatively robust implementations. Due to this, actual crashes are not expected to occur often if at all during the tests.

In addition to normal instrumentation for crashes, code coverage analysis will be performed for the FTP servers Pure-FTPd and Pro FTP Daemon during the test runs.

The tests will consist of all the control commands in FTP, as specified in section 5.3.1 of the FTP specification RFC 959 [79].

Pure-FTPd

Pure-FTPd was chosen because of its popularity and because it is promoted as being designed with security features in mind. [108]

The test setup for Pure-FTPd can be seen in the following list.

- Distribution: Ubuntu 13.04 32-bit, virtualized in Oracle VM VirtualBox 4.3
- Kernel: 3.8.0-31-generic
- Pure-FTPd: version 1.0.20, compiled using "–without-humor, –without-globbing, –with-paranoidmsg" flags with code coverage measurement enabled
- Pure-FTPd configuration: log in enabled, concurrent connection limit 50000, timeout 10 seconds
- Network: VirtualBox host-only network connection

Pure-FTPd is run using close to its default configuration, with the concurrent connection limit raised to a higher value. This is done to prevent false positive verdicts due to not being able to connect to target, if the connections from previous test cases are left open due to anomalies.

Pro FTP Daemon

Pro FTP Daemon (ProFTPD) is, similar to Pure-FTPd, one of the more popular open source FTP servers for Linux platform. Compared to Pure-FTPd, ProFTPD offers more features and more extensive configuration. [109]

Test setup can be seen in the following list.

- Distribution: Ubuntu 13.04 32-bit, virtualized in Oracle VM VirtualBox 4.3
- Kernel: 3.8.0-31-generic
- ProFTPD: version 1.3.0a, compiled using default options with code coverage measurement enabled
- ProFTPD configuration: login enabled, concurrent connection limit 50000, timeout 10 seconds

- Network: VirtualBox host-only network connection

Similarly to Pure-FTPd, ProFTPD is run using default configuration, with the exception of the connection limit and timeout values changed to allow smoother test run in the cases where previous test cases leave their connections open.

4.5.2. Locator/ID Separation Protocol targets

As mentioned earlier, Locator/ID Separation Protocol is a relatively new protocol under development, and because of that there are not many targets openly available. For the purposes of this thesis, the open source implementation of LISP called LISPmob [110] is used.

LISP tests will consist of data plane tests using encapsulated TFTP messages in all four IP version combinations, that is IPv4-in-IPv4, IPv4-in-IPv6, IPv6-in-IPv6, and IPv6-in-IPv4, as well as control plane tests using an encapsulated map request as IPv4-in-IPv4.

LISPmob

LISPmob is an open source implementation of LISP for Linux, Android, and OpenWRT [110]. LISPmob provides the full functionality for both LISP router and LISP mobile node (LISP-MN) on Linux-based systems [111]. For the purposes of this thesis, the focus will be on the router mode. The router mode in LISPmob acts as an xTR, handling the functionality of both an ITR and an ETR [110].

The test setup for LISPmob is presented in the following list.

- Distribution: Ubuntu 12.04 LTS 32-bit, virtualized in Oracle VM VirtualBox 4.3
- Kernel: 3.8.0-29-generic
- LISPmob: v0.4, compiled using default options with code coverage measurement enabled
- LISPmob configuration: router mode
- Network: VirtualBox host-only network connection

LISPmob is configured to run in router mode, as opposed to mobile node, which means it will parse and forward incoming LISP encapsulated data plane packets, as well as respond to encapsulated map requests on control plane.

4.5.3. Session Initiation Protocol User Agent Server

For Session Initiation Protocol testing, the test targets will all be acting as SIP UAS, which means that they will act as servers. Asterisk [112] and SJphone [113] are used as test targets.

The tests are targeting the UAS functionality and the INVITE-CANCEL sequence. The sequence is shown in Figure 8.

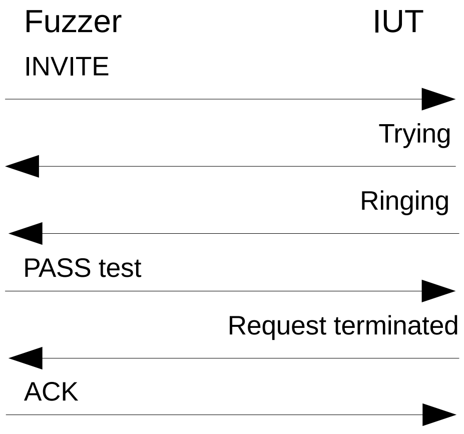


Figure 8. SIP INVITE-CANCEL sequence.

In INVITE-CANCEL sequence, the User Agent Client (UAC), represented by the test suites, starts with an $INVITE$ message, to which the target User Agent Server (UAS) replies with $Trying$ and $Ringing$, confirming that the call is ringing at the other end. Before the call is answered, the UAC hangs up the call with a $CANCEL$ message. The UAS replies with a $Requestterminated$ message, which is ultimately acknowledged by the UAC with an ACK message.

Asterisk

First SIP target is Asterisk [112], configured as a SIP endpoint. Asterisk was chosen because of its high popularity and because it is open source. Asterisk does offer much more functionality besides a simple UAS endpoint, but for this thesis, only the UAS functionality and the INVITE-CANCEL sequence is tested.

- Distribution: Ubuntu 13.04 32-bit, virtualized in Oracle VM VirtualBox 4.3
- Kernel: 3.8.0-29-generic
- Asterisk: 11.4.0
- Asterisk configuration: UAS mode, users with and without authentication, echo test with 5 second delay in answering
- Network: VirtualBox host-only network connection

Asterisk is configured in SIP UAS mode, which means that it is waiting for incoming SIP calls. The configuration has a user that does not need authentication. Also, Asterisk is configured to answer the incoming call with a 5 second delay. In practice, this can be used to test the INVITE-CANCEL sequence, where the test suite cancels the ringing call before it is answered.

SJphone

Another SIP target of choice is SJphone [113]. It is a rather popular, simple VoIP softphone that supports SIP. Being a softphone, it can handle both UAS and UAC functionality, but as with Asterisk, only the UAS functionality is targeted in the tests.

SJphone offers both Linux and Windows versions, but due to some performance problems with the Linux version, namely it randomly ignored new requests, the Windows version was used. More detailed information about the test environment is shown in the list below.

- Operating system: Windows 7 64-bit, native
- SJphone: v.1.65.377a
- SJphone configuration: PC-to-PC SIP mode
- Network: Native Ethernet

The configuration for SJphone is left to defaults, with the use of PC-to-PC SIP call mode.

5. FUZZER DEVELOPMENT AND DEPLOYMENT

The chosen fuzzing frameworks are used to develop a collection of fuzzing *test suites*. The test suites will be deployed and run against a collection of test targets. The used test environment is described in chapter 4.

The test runs are automatized using the provided or self-developed methods for controlling the implementation under test (IUT) when possible. Similar instrumentation methods for different test suites are also chosen when possible, with the main focus on having accurate enough instrumentation to catch the possible crashes.

The used templates for the different fuzzing frameworks are as close to each other as possible, to minimize any effect the template itself might have on the results. In the case of TCF, the valid captures used to create the sequences are recorded from Peach's first valid test case, and further manually edited if needed.

5.1. File Transfer Protocol fuzzing test suites

File Transfer Protocol was chosen as a test protocol because of its simplicity, its popularity especially in the earlier days of the internet, and because it is a relatively old protocol. As stated in section 2.6.1, even though the idea of software testing and fuzzing dates back to a time before the development for FTP started in 1970s, the fact is that only more recently have robustness testing and fuzzing become common practices in addition to the usual conformance testing. This is also why FTP makes an interesting test protocol, as there are many FTP server implementations, ranging from old to new and from complex to extremely simple.

As FTP is relatively simple and it uses the ASCII format for its commands, should also be simple to model using all of the chosen fuzzing frameworks. However, FTP does pose one problem: the data connection. As explained in section 3.6.1, two different TCP connections are used. The first is for control, e.g. sending all of the FTP commands, and the second connection is used for the actual data transfers [79]. This has the potential of making the testing problematic since most frameworks are designed with single connection protocols in mind; suddenly having another TCP socket open is a challenge. For the scope of this thesis, only the control connection is fuzzed, and the data connection is ignored.

5.1.1. Peach

A Peach Pit file that defines all of the used FTP commands and command sequences was written. For the FTP commands that need to be executed after a successful log in, a valid log in sequence was run before the command itself. That is, valid $USER$ and $PASS$ commands were sent before the actual anomalized command, and a valid $QUIT$ was sent at the end.

As instrumentation, Peach's LinuxCrashMonitor and Process monitors were used. As the target resided on another machine over the network, remote Peach Agent was used to run the actual monitors. Process monitor was also used to control the execution of the IUT, starting and restarting it as needed.

5.1.2. Sulley

All of the tested FTP control commands were first defined as Sulley requests. The commands were connected together to form a state graph which Sulley will then walk through while testing, as described in section 3.3.1. Similar to Peach, all of the anomalized control commands not related to log in were connected to a valid log in sequence, $USER$ and $PASS$ commands, in order to first pass the log in phase while testing.

Again similar to Peach, remote process monitoring instrumentation was used for the test runs. Sulley's Procmon was run remotely and it was used to both monitor the health of the target as well as start and stop it.

5.1.3. Traffic Capture Fuzzer

As mentioned in section 3.5, TCF uses a valid network capture as input, parsing the protocol template from that using Wireshark's dissectors. To be on the same line with the other frameworks, the valid capture is taken from Peach's valid test case. This is done separately for all of the different FTP control commands. Due to this, all of the sequences will automatically also include the log in and disconnect commands.

The main instrumentation method used with TCF is the valid-case instrumentation. In addition to valid-case instrumentation, the developed process monitoring script, defined in section 3.5.5, is used to control the execution of the target.

5.2. Session Initiation Protocol fuzzing test suites

Session Initiation Protocol (SIP) is a more complex protocol than FTP, but it is still formatted as ASCII text, which should make the templating straightforward. The testing scope was limited to the INVITE-CANCEL sequence. As mentioned in section 3.6.2, many SIP implementations also require the session information to be present in the payload as Session Description Protocol (SDP) data. For these tests, the SDP data is also included in the templates and will be anomalized together with the actual SIP headers.

A thing to note about SIP is that a special Call-ID value is used to distinguish SIP calls from each other. The Call-ID is a part of the SIP header and it should remain the same for every message during the same call, and also be unique to that one call, meaning that the following calls should have a different Call-ID. Otherwise, if the Call-ID is not changed, the target implementation might end up rejecting the new call. This means that some dynamic functionality for the developed test suites is needed.

5.2.1. Peach

Originally, the $DataModel$ elements for representing SIP messages in the Peach Pit file were created in extreme detail, with each segment of the commands and tokens (spaces, commas, other token-like parts) individually defined as seen in Listing 11.

```
<DataModel name="DM_SIP_Invite">
  <String name="command" value="INVITE"/>
  <String value=" " token="true"/>
  <String name="protocol" value="sip"/>
  <String value=":" token="true"/>
  <String name="username" value="102"/>
  <String value="@" token="true"/>
  <String name="IP" value="172.16.0.101"/>
  <String value=" " token="true"/>
  <String name="versionSIP" value="SIP"/>
  <String value="/" token="true"/>
  <String name="versionNumber" value="2.0"/>
  <String value="\r\n" token="true"/>
...
```

Listing 11. Example element from maximally detailed Peach Pit file

It was soon noted that while this sort of definition leads to more accurate anomalies, it also drastically increases the number of test cases, quickly reaching the point where running that many test cases is no longer feasible. Due to this, the template was simplified. The change in the template does not affect the actual structure of the outgoing messages, only anomaly generation. An example of the simplified definition of the same example as above is shown in Listing 12.

```
<DataModel name="DM_SIP_Invite">
  <String name="command" value="INVITE "/>
  <String name="user" value="sip:102@172.16.0.101 SIP/2.0"/>
  <String value="\r\n" token="true" mutable="false"/>
...
```

Listing 12. Example element of a simplified element definition in Peach

In the simplified version, many of the different smaller elements had to be combined into a single element. The rest of the elements were defined in similar fashion.

The test sequence was created based on a valid INVITE-CANCEL sequence, where the test suite first sends $INVITE$, awaits for $Trying$ and $Ringing$ from the IUT, then replies with $CANCEL$, receives $Requestterminated$ confirmation, and then finally replies with ACK.

For dynamically changing the Call-ID, a Python ScriptFixup is used to generate the changing portion of the Call-ID.

Instrumentation for Peach is the same as with FTP for Asterisk: LinuxCrashMonitor and Process monitor for checking the target processes for crashes as well as controlling their execution. For SJphone, LinuxCrashMonitor could not be used for obvious reasons.

5.2.2. Sulley

The requests and sequences in the test suite created using Sulley were modeled with as much detail as possible considering the anomalization. An example of the template for $INVITE$ is shown in Listing 13.

```
s_initialize("INVITE")
s_string("INVITE")
s_delim(" ")
s_string("sip")
s_delim(":")
s_string("102")
s_delim("@")
s_string("172.16.0.101")
s_delim(" ")
s_string("SIP")
s_delim("/")
s_string("2.0")
s_delim("\r\n")
```

Listing 13. An example of a partial SIP $INVITE$ message using Sulley

The rest of the message, as well as the other messages, were defined in similar, extremely detailed manner. Due to the Sulley engine creating less anomalies per field compared to Peach, the test case amount did not turn out to be a problem with the test suite developed using Sulley.

For instrumentation, Sulley's Procmon was run remotely and it was used to both monitor the health of the target as well as start and stop it.

5.2.3. Traffic Capture Fuzzer

TCF used once again the valid capture from Peach. Some manual tweaking was needed to properly mark the Call-ID value in the sequence file for outgoing messages. The Software Development Kit (SDK) addition to TCF was used to dynamically change the Call-ID value between test cases.

The main instrumentation method used with TCF was the valid-case instrumentation. In addition to valid-case instrumentation, the developed process monitoring script, defined in section 3.5.5, was used to control the execution of the target.

5.3. Locator/ID Separation Protocol fuzzing test suites

Locator/ID Separation Protocol (LISP) is much more complex protocol than FTP or SIP, including the use of IP and UDP headers in the inner header of the LISP packets. As the inner LISP packet is carried within a regular outer UDP packet, and all the interesting LISP data is in the inner LISP header and payload, templating the outer header can be left out of the model, letting the UDP injectors found in all of the frameworks handle that part. That is, only the inner header (IH) and the payload will be templated, see section 3.6.3 for details of the LISP encapsulation.

Peach and Traffic Capture Fuzzer are capable of injecting raw Ethernet frames as well, which would allow the whole packet to be templated, but anomalizing the outer header (OH) would be targeting the UDP/IP stack of the underlying operating system rather than the target process itself. Also, as mentioned before, Sulley does not have any injectors capable of transmitting raw Ethernet frames.

The data plane tests included encapsulation using all of the supported IP version combinations, that is IPv4-in-IPv4, IPv4-in-IPv6, IPv6-in-IPv4, and IPv6-in-IPv6, using a simple TFTP read request as the actual payload in the inner IP packet. The control plane tests were done using an encapsulated Map request which is inserted as the inner payload. In both data and control plane tests, the payload was also anomalized by the developed test suites.

5.3.1. Peach

In LISP data plane tests, the inner payload is a complete IPv4 or IPv6 packet. Peach offers a fairly good environment for templating it. For example, the $Flag$ elements in Peach can be used to represent the one bit flags in the LISP header, as well as using the $Number$ element to represent numeric values in different formats (decimal, hex, binary), as seen in Listing 14.

```
<Block name="LISP">
 <Flags name="LISPFlags" size="8">
  <Flag name="N" position="0" size="1" value="1"/>
  <Flag name="L" position="1" size="1" value="0"/>
  <Flag name="E" position="2" size="1" value="0"/>
  <Flag name="V" position="3" size="1" value="0"/>
  <Flag name="I" position="4" size="1" value="0"/>
  <Flag name="Flags1" position="5" size="1" value="0"/>
  <Flag name="Flags2" position="6" size="1" value="0"/>
  <Flag name="Flags3" position="7" size="1" value="0"/>
 </Flags>
 <Number name="Nonce" size="24" valueType="hex"
         value="0xff 0xff 0xff"/>
 <Number name="Locator-status-bits" size="32" valueType="hex"
         value="0x00 0x00 0x00 0x00"/>
</Block>
```

Listing 14. LISP header template in Peach

The LISP template for data plane tests includes the LISP header, inner IP header, and the payload consisting of an UDP header and the simple TFTP read request packet. The outer IP and UDP headers are handled by Peach's UDP injector.

The encapsulated Map requests on the control plane are fairly similar. A LISP header is followed by an IPv4 header, UDP header, and the LISP Map request itself as payload. Again, Peach's UDP injector will handle packaging the message into a standard UDP/IP packet.

Instrumentation for LISP tests is done with LinuxCrashMonitor and Process monitor for checking the target process for crashes and restarting it if needed.

5.3.2. Sulley

LISP, even when templating only the outer headers and payload and using a normal UDP injector, is a problem for Sulley. Templating LISP requires the use of many very short fields, for example the one-bit flags, five-bit IRC value in Map request, and more.

As mentioned in section 3.3.1, Sulley does not support bit-sized primitives as elements in the template. They are always padded to the nearest byte, which makes it impossible to write a proper template for LISP with Sulley.

Due to this, Sulley had to be excluded from the LISP tests.

5.3.3. Traffic Capture Fuzzer

For Traffic Capture Fuzzer the used valid captures were once again taken from the valid test case from the test suite developed with Peach, one for each different packet type. In TCF, the Application playback replay type was used, meaning that only the application layer data, onward from the LISP header, is added to the sequence. With Application playback, TCF will use its own UDP/IP injectors to handle the outer headers, similar to Peach.

In Listing 15 is an example of an imported IPv4-in-IPv4 LISP data plane packet and its LISP header part.

```
<label t="lisp_data">
 <label t="flags">0x01</label>0xff 0xff 0xff
  0x00 0x00 0x00 0x00
</label>
```

Listing 15. Captured LISP header in TCF sequence

In this example, the LISP Nonce and Locator-status-bits fields have not been identified correctly by the Wireshark protocol dissector during the import and shown only as raw data. This can sometimes happen if the dissector is not perfect and makes mistakes. The sequence can be manually edited to reflect the correct structure, as seen in Listing 16.

```
<label t="lisp_data">
 <label t="flags">0x01</label>
 <label t="nonce">0xff 0xff 0xff</label>
 <label t="locator-status-bits">0x00 0x00 0x00 0x00</label>
</label>
```

Listing 16. Edited LISP header in TCF sequence

This will result in better anomalization and anomaly placements.

The developed process-based instrumentation script was used to detect crashes and control the test target. For the data plane Map request tests, also valid-case instrumentation was used.

6. RESULTS

The test runs were executed in controlled environments defined in the previous chapters. Default anomalization settings were used for all of the developed test suites in order to prevent unnecessary variance in the results due to configuration. The results were recorded and analyzed.

Interpreting the results can be a complex task. For example, when looking at the statistics for found crashes, a high number of failures represents a definite problem in the target, but defining whether this is because the fuzzer is exceptionally good or the target exceptionally bad is a more difficult task. [24 p. 106-113]

As mentioned briefly in section 4.3.1, code coverage has a similar problem. The code coverage will show how much of the executable code is covered during the test run, but a low coverage percentage does not necessarily mean that the fuzzer was not effective; it could simply mean that the reachable functionality from the used attack vector only covers a small part of the code.

The code coverage measurements were performed for the File Transfer Protocol targets Pure-FTPd and ProFTPD and Locator/ID Separation Protocol target LISPmob. The coverage measurements were not performed for Session Initiation Protocol due to the protocol being so extensive, since the scope of SIP tests would not have revealed reasonable results. The Damerau-Levenshtein distance measurements were performed for the SIP protocol $INVITE$ message due to it being both in clear-text and long enough to give proper results.

For the purposes of this thesis, all of the test targets will be run in equal conditions, and the attack vectors will also be the same for all fuzzers. This ensures that a quantitative comparison can be made between them using the gathered results.

6.1. File Transfer Protocol

The developed test suites were tested against a selection of FTP server implementations. As mentioned in section 5.1, the separate data connection poses difficulties when fuzzing FTP. In these tests, all of the commands, including those that require a new data connection, were covered, but no separate data connections were made.

As most FTP commands must be preceded with a valid log in phase ($USER$ and $PASS$ commands) in order to drive the target's internal state machine to a proper state, and optionally followed by a $QUIT$ message for valid disconnection, the templates included the $USER$, $PASS$, and $QUIT$ commands as part of the sequences.

Anomalizing the log in ($USER$ and $PASS$ commands) and disconnect ($QUIT$ command) phases were excluded from the test cases where possible. With Sulley and Peach this is straightforward since the anomalization of certain parts of the templates and sequences can be disabled, but for Traffic Capture Fuzzer a similar feature was still under development at the time of writing this thesis. However, as all of the fuzzing engines including TCF anomalize only one message at the time, excluding the combination anomaly cases, this did not manifest into a problem from the point of view of the target's state machine, because when anomalizing the control command after logging in, TCF would add an anomaly only into that message and leave the preceding $USER$ and $PASS$ commands untouched.

6.1.1. Performance

The developed test suites were executed against Pure-FTPd and ProFTPD. The results are shown in Table 7.

Table 7. Instrumentation results from FTP tests

Fuzzer	Test cases	Crashes in Pure-FTPd	Crashes in ProFTPD
Peach	308074	0	0
Sulley	50624	0	0
TCF	3135863	0	0

As suspected before the test runs, both targets proved to be extremely robust and no crashes were found using any of the test suites.

The amount of test cases generated by the different test suites and their fuzzing engines varied greatly. While Sulley generated slightly over 50000 cases altogether, Peach generated over 300000 test cases, which was again increased by roughly a factor of ten by TCF to reach over 3 million test cases. There are couple of factors contributing to the difference.

Due to the missing feature mentioned above for disabling anomalization for certain parts of the sequence, TCF anomalized the whole test sequence, including $USER$, $PASS$, and $QUIT$ commands, for all the individual sequences for all the FTP commands. This means that, in the case of FTP where the log in and disconnect use three of the usual four messages sent, from the total cases run across the different sequences with TCF roughly 75 $\%$ are duplicates, bringing the estimated amount of unique test cases close to 700000-800000.

In addition to this, compared to Sulley, the anomalies Peach and TCF create are more extensive. As discussed earlier in section 2.6.6, there are many different anomaly types. Within each type, there are different variations of the anomaly. Peach and TCF create more of these variations for each type of anomaly and for each part of the template. For example, all of the engines in the frameworks make use of field level overflows by simply repeating a value inserted there. Peach and TCF have more variation to this repetition compared to Peach, thus making up more test cases.

The test suites were also tested by running the anomalized log in sequence, $USER$, $PASS$, and $QUIT$ commands, and measuring the test run speed. The results from the runs against ProFTPD are shown in Table 8. In these tests, no additional instrumentation was used. This was to minimize the overhead instrumentation might add to the running speed and to ensure comparable test results.

Table 8. FTP performance results for log in sequence against ProFTPD

Framework \ Results	Test cases	Elapsed time	Test cases per second
Peach	21490	02:22:24	2.51
Sulley	5474	00:03:46	24.22
TCF	76768	00:38:18	38.41

Again, the results indicate that there is a considerable difference in the amount of generated test cases even though the template is essentially the same for all of the test suites. Sulley is generating considerably fewer test cases than the other two.

The running speed of the test suite developed using Peach is only 2.51 test cases per second, considerably slower than the two other suites. Sulley achieved much better results with just over 24 test cases per second, and that, combined with Sulley aiming to generate more concise test cases, makes test suites developed using Sulley extremely quick to run. However, TCF achieved the best running speed by a wide margin, averaging almost 40 cases per second. The fast running speed and a greater number of test cases assure wider input space coverage in a manageable run time.

6.1.2. Code coverage

Code coverage was measured for the test targets after running all of the test sequences using default anomalization engine settings, with the exception of excluding duplicate anomalization as mentioned before. The results can be seen in Tables 9 and 10.

Table 9. Code coverage for Pure-FTPd

Framework \ Coverage	Line	Function	Branch
Peach	9.0 %	18.3 %	4.6 &
Sulley	1.6 %	4.1 %	0.9 %
TCF	9.0 %	18.3 %	4.6 &

Table 10. Code coverage for Pro FTP Daemon

Framework \ Coverage	Line	Function	Branch
Sulley	15.5 %	22.3 %	9.2 %
Peach	15.8 %	22.7 %	9.4 %
TCF	15.7 %	22.6 %	9.3 %

All of the suites were using templates that were as identical as possible, which should guarantee that the test cases hit the same attack surface. The reason why the overall coverage is low is that both test targets implement more features than the basic FTP commands tested here, for example different daemonization options and sophisticated user and sharing controls, which were not configured in the targets, and the code might also have parts not accessible through the external attack surface.

From the results it can be clearly seen that the while Peach and TCF reached similar code coverage results for both Pure-FTPd and ProFTPD, Sulley was struggling against Pure-FTPd with its much narrower coverage.

Against ProFTPD, the three test suites reached almost identical coverage results. Even the suite developed using Sulley reached the same coverage despite generating only a fraction of the amount of test cases compared with the two other frameworks (as seen in Table 7). As mentioned before, a majority of the total amount of test cases generated with TCF are duplicates, which also supports the notion that a smaller set of test cases would be enough in order to reach similar code coverage results.

6.2. Session Initiation Protocol

Session Initiation Protocol (SIP) tests were performed using the INVITE-CANCEL sequence, where the User Agent Client (UAC), represented by the test suites, starts with an $INVITE$ message, which the target User Agent Server (UAS) replies with $Trying$ and $Ringing$, confirming that the call is ringing at the other end. The UAC then hangs up the call with a $CANCEL$ message, replied by the UAS with a $Requestterminated$ message finally acknowledged by the UAC with an ACK message.

With Peach, the problem with fuzzing SIP was the amount of generated test cases combined with the slow running speed, as discussed in section 5.2.1. With a very accurate template, which should help generate better anomalies, Peach generated over 1.7 million test cases for the INVITE-CANCEL sequence. This, combined with the average running speed of 0.8 test cases per second with instrumentation after the first 20000 test cases, caused the estimated run time for the fuzzer to be over 26 days, which is completely unfeasible. Due to that, the Peach template was simplified and the number of test cases reduced to a number that Peach could run in a reasonable time.

Sulley and Traffic Capture Fuzzer did not have similar problems so their templates were kept as accurate as possible. The final results for SIP runs against Asterisk can be seen in Table 11.

Table 11. Instrumentation results from SIP INVITE-CANCEL tests against Asterisk

Results / Framework	Test cases	Crashes (reproducible)	Crashing groups
Peach	220659	0	0
Sulley	175636	0	0
TCF	166099	4 (0)	2*

After simplifying Peach's template, the generated test case amounts are fairly similar in all of the test suites.

The test suite developed with TCF was the only one to find crashes in Asterisk. The crashes were not reproducible using single test cases, nor were there any indications of a reason in the IUT logs. A full re-run revealed further crashes, but with different test cases, which leads to an assumption that the crashes are most likely caused by resource exhaustion, or a race condition that could not be resolved, that only happens over a longer time frame.

The test suites were also executed using SJphone as a target. Compared to Asterisk, SJphone is a much simpler implementation, a basic software phone. It was also noted from the start that the performance of SJphone was much worse than that of Asterisk and the testing was considerably slower. As an older release, SJphone was expected to be less robust than the Asterisk implementation tested earlier. The results for the test runs are shown in Table 12.

Table 12. Instrumentation results from SIP INVITE-CANCEL tests against SJphone

Results / Framework	Test cases	Crashes (reproducible)	Crashing groups
Peach	220659	0	0
Sulley	175636	0	0
TCF	152545	169(129)	16*

Surprisingly, Peach did not manage to crash SJphone with INVITE-CANCEL sequence. One factor in this is the problem mentioned earlier with Peach's ScriptFixup, which was used to generate the SIP Call-ID for each call. For an unknown reason, Peach's engine did not run the ScriptFixup for each test case, which resulted in using the same Call-ID for many calls, which SJphone often ended up rejecting and caused problems for some test cases where the anomaly was inserted into one of the latter messages, $CANCEL$ or ACK. However, a majority of the test cases was generated for the $INVITE$ message, since it has the most data, and, confirmed by analyzing SJphone's log files, all of them were parsed and none of them managed to find crashes either.

Similar to Peach, Sulley did not find any crashes from SJphone.

TCF found 129 reproducible crashes from SJphone. There were 16 different test groups that were causing crashes. Each test group is designed to test a different element in the messages. Many of the crashes were found with anomalies in the Session Description Protocol (SDP) information sent as payload in the $INVITE$ message. Crashes were also found with anomalies in the actual SIP headers in all of the three messages ($INVITE$, $CANCEL$, and ACK).

In addition to this, when using valid-case instrumentation it was noted that on many occasions more than one instrumentation rounds were needed before the IUT responded. This may indicate that although the IUT did not crash completely, it was not able to respond properly right after an anomaly, possibly pointing towards resource exhaustion problems.

Due to the poor performance of SJphone, which was a limiting factor in itself, the test suite performance results from the runs were executed with Asterisk as a test target. The results can be seen in Table 13. In the tests, 500ms timeout value was used for received data messages in the test suites.

Table 13. Performance results from SIP INVITE-CANCEL tests against Asterisk

Results / Framework	Test cases	Run time	Test cases per second
Peach	220659	31:15:09	1.96
Sulley	1756363	00:17:05	171.35
TCF	166099	00:25:23	109.03

Compared to FTP, which runs on top of TCP, the SIP tests were run on top of UDP. UDP should be faster on the transport level because there is no three-way handshake or closing sequence like there is in TCP. The test sequence is also a little shorter in SIP compared to FTP, which should also make the testing faster.

The results from Sulley and TCF are in line with that assumption. Both scored faster running speed results with SIP compared to FTP. Sulley's speed of over 170 test cases per second is clearly the fastest with SIP. The way Sulley walks through the test graph helps since it does not always need to wait for a response from the target for the first test cases in the sequence. TCF also reached a little over 100 test cases per second, which is still quite fast.

Peach was extremely slow compared to the two others, averaging just under two test cases per second. As noted above, with instrumentation the 20000 first cases averaged even less, under one test case per second. When analyzing the logs and captures, it is clear that this is caused by Peach hitting the timeout value on almost every test case. This seems to be because the IUT, Asterisk, answers with error messages to most anomalized messages sent by the test suites. The suite made with Peach does not react to unexpected messages in any way, it simply waits until the timeout. TCF and Sulley both handle the incoming unexpected messages properly and move to the next case if an unexpected message has been received after an anomaly. As mentioned in section 3.1.1, the behavior of the IUT after an anomaly is not specified and thus it is not important.

6.2.1. Damerau-Levenshtein distance

While testing SIP, the anomalies made to the INVITE messages were recorded and compared to the valid INVITE using normalized Damerau-Levenshtein distance. The average distances are shown in Table 14.

Table 14. Normalized Damerau-Levenshtein distances for SIP INVITE anomalies

Framework	Average
Peach	0.311
Sulley	0.689
TCF	0.380

On the normalized Damerau-Levenshtein scale, 0 means the compared messages were equal and 1 that there were no similarities.

From the results it can be clearly seen that while Peach and TCF scored similar results, Sulley's average distance stands out as almost double to that of Peach and TCF. This means that the anomalized messages Sulley's engine created are much further from the original valid case and thus are less likely to be accepted by the IUT. Possible reasons for the higher distance could be the use of combination anomalies, long overflow anomalies, or in general bad mutation anomalies changing too big of a portion of the message at once.

Peach and TCF scored a much lower average distance, which points to smaller, more targeted anomalies, that are more likely to be accepted by the IUT.

This was also observed during the actual test runs against SJphone. The $INVITE$ messages from test suite using Sulley only rarely managed to open up new calls due to the anomalies changing the message too much, whereas the test suites made using Peach and TCF opened calls on a very high number of test cases.

6.3. Locator/ID Separation Protocol

The two different planes in Locator/ID Separation Protocol, data and control plane, were both tested using different test case sequences. For data plane tests, normal LISP encapsulated TFTP packets were used. For control plane, the tests were done using encapsulated LISP map requests.

Process-based instrumentation was used for checking the health and starting and restarting the LISPmob process on the target machine. With Traffic Capture Fuzzer, also valid-case instrumentation was used for the control plane tests.

As mentioned earlier, Sulley does not support low-layer protocols due to a problem with templates with bit-sized fields, which means LISP and the needed IP and UDP headers in the inner headers can not be defined. These tests were run only using Peach and TCF.

The target used in all the LISP tests was LISPmob v0.4 running in router mode.

6.3.1. LISP Data plane

The data plane tests were run using all combinations of supported IP versions, that is IPv4-in-IPv4, IPv6-in-IPv4, IPv6-in-IPv6, and IPv4-in-IPv6. The results can be seen in Tables 15, 16, 17, and 18.

Table 15. Instrumentation results from LISPmob data plane encapsulation tests, IPv4-in-IPv4

Results / Framework	Test cases	Crashes (reproducible)	Crashing groups
Peach	83935	0	0
TCF	27159	0	0

Table 16. Instrumentation results from LISPmob data plane encapsulation tests, IPv6-in-IPv4

Results / Framework	Test cases	Crashes (reproducible)	Crashing groups
Peach	84737	0	0
TCF	27085	0	0

Table 17. Instrumentation results from LISPmob data plane encapsulation tests, IPv6-in-IPv6

Results / Framework	Test cases	Crashes (reproducible)	Crashing groups
Peach	84737	0	0
TCF	27085	0	0

Table 18. Instrumentation results from LISPmob data plane encapsulation tests, IPv4-in-IPv6

Results / Framework	Test cases	Crashes (reproducible)	Crashing groups
Peach	83045	0	0
TCF	27141	0	0

Neither of the two test suites found any crashes from LISPmob using the data plane sequences. There is some difference in the amounts of generated test cases, where the test suite using Peach generated roughly three times as many test cases as TCF using a similar template.

6.3.2. LISP Control plane

LISP control plane was tested with encapsulated map request packets. As mentioned earlier, with Sulley it is not possible to create accurate enough templates for lower layer protocols, so Sulley was not part of the test runs. The results for Peach and TCF can be seen in Table 19.

Table 19. Instrumentation results from LISPmob control plane tests

Results / Framework	Test cases	Crashes	Reproducible	Crashing groups
Peach	104847	4644	4534	15
TCF	28410	4195	4195	20*

From the results it can be clearly seen that both fuzzers managed to find a high number of crashes. However the high number of crashes found does not necessarily mean that the IUT has that many bugs; with high probability the underlying bug count is much lower and the different anomalies triggered same bugs multiple times. The amount of different crashing test groups indicates this as well.

While the number of overall crashes is high, the number of different test groups, essentially tested elements, is much lower. For example, out of the 4534 reproducible crashes found by Peach's test suite, 4208 were found by anomalizing the length value in the inner UDP header. The fact that a single test group with anomalies targeting a single element found a high number of crashes is a fair indication that there is problem in the code handling that part of the message in the IUT. This does not, however, mean that there are necessarily a high number of bugs; it can only be a single bug reproducing with multiple anomalous inputs.

Checking the amount of different test groups is a bit tricky in test suites using TCF since the fuzzer engine in TCF creates a so called *extended* group, which holds the test cases created by mutation, see sections 2.6.5 and 3.5, which is the vast majority of the test cases, and inside the extended group there is no detailed test case information available.

Even though the number of found crashes is similar with both test suites, the one using Peach needed almost four times the amount of test cases to trigger the crashes. The test suite using TCF reached a hit ratio of nearly 0.147 crashes per test case, meaning roughly one crash per seven test cases. For the other test suite, the hit ratio for reproducible crashes was considerably lower, 0.043, meaning a crash every 23 test cases on average.

6.3.3. Coverage

In addition to the normal instrumentation for crashes, code coverage was measured for the LISP runs, including both data and control plane runs. The results are shown in Table 20.

Table 20. Code coverage for LISPmob lispd

Framework \ Coverage	Line	Function	Branch
Peach	27.0 %	40.8 %	18.0 %
TCF	37.9 %	54.6 &	25.6 %

The results are in a clear favor of the test suite using TCF. It reached over 10 percentage units higher line coverage than the test suite using Peach, and almost 14 percentage units higher function coverage.

6.4. Remediation and reporting

With Peach, the remediation phase is highly dependent on the instrumentation method that is used with the test suite during the test run. As Peach does not have any dedicated remediation nor reporting methods, the only tools and available information for remediation are the logs and result files from the test runs. However, because of the wide collection of different instrumentation methods offered by Peach, the results can often offer extended knowledge of the problems. The LinuxCrashMonitor used in most of the performed tests automatically gathered *crashdumps* from the IUT using the GDB debugger. For Windows, there is a similar monitor called Windows Debugger. For a software developer, this information can be almost as useful as being able to reproduce the problem in a debugger later on. The reproducing is also possible with Peach by simply having a copy of the Pit file and running the test case with the same index. One big problem with Peach is that there is no clear information given about the anomaly itself. The only way to see the actual anomalized messages is to capture the network traffic.

Sulley does not have any dedicated remediation or reporting either. The Sulley test suites gather some log files for each test run, but those do not include any useful information about the anomalies or responses from the target. For instrumentation, Sulley's Procmon is very similar to the debug monitors used by Peach. It records the data from the crash to a crashbin file. With a help of a built-in Crashbin explorer script, the information stored in the file, including the crashdumps, can be further analyzed. Again, for a seasoned tester or developer this information gives a fair idea where the actual crash happened in the code and whether it can be exploited by an attacker.

Traffic Capture Fuzzer offers a few ways for remediation and reporting the results. A special *remediation package* can be created which includes the test results and log files, settings, used test sequence, and a brief report of the test run. The package can be used to replicate the test run with another instance of TCF. The log files can also be used to look up the detailed information on the used anomalies. The remediation package does not include any crashdumps or other additional information similar to Peach and Sulley, because the default instrumentation methods in TCF do not gather anything like that from the target. If, however, additional external instrumentation is used, the output of that will be included in the logs. As mentioned in section 3.5.6, the test run information, including the test case descriptions detailing the anomalies, can be also exported into easily understandable reports. All in all, for detailed crash information, the users of TCF must rely to external methods.

6.5. Development and deployment complexity

Table 21 shows the lines of code per test suite, excluding all comments and empty lines used for formatting purposes. The value for Peach SIP tests is for the simplified template. The value for the original template was 450 LoC.

Table 21. Lines of code per test suite

Framework \ Protocol	FTP	LISP	SIP
Peach	1898	394	204
Sulley	143	-	377
TCF	5079	276	673

The difference between the lines of code for each test suite is formidable. Sulley uses the fewest LoC for the same effect, due to the very efficient way of defining both the message templates and the test sequence. In Peach, the templates for the messages are defined only once, but the used $StateModel$ elements defining the test sequences and $Test$ elements defining the test runs include a great deal of repeated information, which makes the definition longer, especially for FTP that has such a high number of command sequences to define. Traffic Capture Fuzzer has a similar problem than Peach, with the addition that each sequence contains also the template definition for all of the messages in the sequence. This is mostly due to the fact that while the templates and sequences for a single protocol in Peach and Sulley are part of the same test suite, having to import multiple captures in TCF essentially creates a new and complete test suite each time. However, it should be noted that TCF does most of this work automatically in the import phase.

Also the configuration of the test run, meaning target definitions, instrumentation and other settings, are embedded in the templates in the test suites using Peach and Sulley. In TCF, this configuration is separate, and not part of the measurements here.

Documentation is a big part of a development process. Peach has online documentation listing all of the different building parts of the framework with some simple examples. Unfortunately, the documentation was found to have many parts that are out of date or simply missing, bringing the overall quality and usefulness of the documentation down. For example, some features, like the ScriptFixup, were only present in the few examples shipped with Peach and there was no mention of them in the online documentation.

Sulley provides very simple documentation which goes through the test suite development using simple examples. In addition, an API documentation can be generated using Epydoc [114] for all the classes and methods available in Sulley. The main problem with Sulley's documentation is that having just a plain API documentation requires a strong knowledge of both the used programming language, Python, and the framework itself.

In the case of TCF, most of the available documentation is built into the tool itself. Some additional information is distributed in the form of a so-called Solution Note, which is a walkthrough of TCF usage with simple examples. The tool itself has fairly comprehensive documentation on the overall usage, but the documentation regarding editing the sequence file, that is used as the template, is extremely limited. While the import function that takes in a valid capture and creates the sequence based on the traffic does most of the work creating the template, further manual editing is often needed and the documentation regarding that is scarce.

The Software Development Kit part has its own Solution Note, accompanied with an API documentation. Again, similar to the case with Sulley, the API documentation alone requires fair knowledge of both the Java programming language as well as the architecture of TCF itself, which can prove overwhelming at first.

Deployment into actual use is fairly similar with all of the frameworks. They are all built as software components and can be run in Windows and Linux. Sulley and Peach can also be run in OS X, but TCF does not support that. The main difference arises when setting up the instrumentation. Since Peach and Sulley only have software-based instrumentation methods capable of giving test case verdicts, those methods can only be deployed on target machines with a compatible operating system and where the actual IUT is a controllable software process. TCF has an advantage here because of its valid-case instrumentation and the capability to use any custom external instrumentation method, which makes it independent regarding the target environment.

7. DISCUSSION

In this chapter the obtained results are assessed in further detail. Also some thoughts are given for the known limitations in the work, and how the work could be developed further.

7.1. Assessing the results

Traffic Capture Fuzzer was found to be the most effective in terms of found crashes. TCF found crashes from Asterisk, SJphone, and LISPmob, as seen in Tables 11, 12, and 19. Peach found a higher number of crashes from LISPmob, but with a significantly higher test case count, bringing the actual crash per test case ratio to a much lower value. TCF was also the only one to find crashes from Asterisk and SJphone, even though the crashes from Asterisk were not reliably reproducible.

On a run-time performance level, TCF and Sulley were the clear favorites. The test suites using Peach were, in most cases, extremely slow compared to the others. This compared with the fact that Peach's engine tends to generate more test cases than the others, especially when making as accurate templates as possible with a high number of elements, makes utilizing Peach a tough choice when high performance and quick test run time are important.

The code coverage measurements showed that Peach and TCF can achieve very similar results in code coverage at least for text-based protocols such as File Transfer Protocol, as was shown in Tables 9 and 10. With a more complicated protocol, coming down to bit-level elements, TCF managed a better coverage, as was seen in the Locator/ID Separation Protocol test results in Table 20. Sulley was at the same level with the two others on one of the two FTP tests, but fell far behind on the other, not really putting up any competition there.

One of the reasons for Sulley's poor code coverage could be the high Damerau-Levenshtein distance of the test cases, meaning that generated test cases were very different compared to the valid case, probably leading to them being outright rejected by the targets in most cases. The test cases generated with Peach and TCF had a lot lower Damerau-Levenshtein distance, almost half of the value of Sulley's test cases.

The development complexity was a bit harder to measure quantitatively since the development process is different for all of the frameworks. The measured lines of code per test suite shown in Table 21 indicates how complex the templates itself are and how well designed their processes and features are. Sulley was the clear front runner here, providing very convenient ways of defining both the message templates and especially the test case sequences. For protocols with multiple messages and test case sequences, Peach requires a lot of repetition. The same is true for TCF due to the fact that different test sequences are essentially different test suites in TCF.

As part of the development process, the framework documentation also played a key role. In this sense, all of the frameworks could have done better. None of them offered very comprehensive documentation and parts of the development had to be done by simple trial and error method.

TCF claims the pole position in overall usability. It is the only one of the three frameworks that provides a graphical user interface, and the automatic sequence creation from a network capture is a superior feature. With the help of Netzob, developing a template and sequence can be done from an important capture and exported to Sulley, but the process is not nearly as automated and seamless as it is in TCF.

From a product quality point of view, both Peach and Sulley fell behind Traffic Capture Fuzzer. TCF, as a commercial software product, is much more polished, robust, and user-friendly. This is, of course, to be expected from a commercial product. Both Sulley and Peach were found to have usage hindering bugs, like the randomly working ScriptFixup in Peach, and missing essential features and functions, like the lack of network layer injectors in Sulley and the inability to include elements smaller than one byte into the templates.

Overall, between the tested frameworks, Traffic Capture Fuzzer performed the best. There were no real problems at any point during the development or test execution. Peach did not find as many crashes as TCF, but it still proved to be effective. The built-in injection and instrumentation methods were also the most extensive. The major problem with Peach was the runtime performance, which was extremely poor compared to the others. Sulley's runtime performance was better, but it lacked in effectiveness, not finding any reproducible crashes. This was likely due to the narrower test material.

At the same time, it was also further proved that fuzzing is an effective way of finding vulnerabilities and flaws in software. The developed fuzzing test suites found a great deal of crashes in various targets. This goes to show that fuzzing, even when not using the most effective fuzzer, is always worthwhile.

7.2. Known limitations

Due to the missing injectors in Sulley, the chosen test protocols were mainly application layer protocols. This somewhat limits the results as there were no tests for transport or network layer protocols to see how the fuzzing engines would have performed in that situation. However, the Locator/ID Separation Protocol tests gave some indication, because they include a complete IP packet in the payload.

Some limitations exist also with the used Damerau-Levenshtein distance. The distance measurement requires a string of text, which rules out LISP. Also, due to the shortness of File Transfer Protocol messages, the distance measurement could only be performed reliably for Session Initiation Protocol.

7.3. Future development

In this thesis, only a few of the most popular fuzzing frameworks and protocols were covered. To get a more complete understanding of the effectiveness of fuzzing frameworks, more protocols could be fuzzed and the amount of test targets increased, which would give more test data to analyze.

Adding more fuzzing frameworks is also an intriguing possibility, for example employing more commercial frameworks (both template-based and model-based) against the available free frameworks. In the case of Peach, which also has a commercial version with more advanced fuzzing engine available, testing the free and the commercial version side by side could give a good estimate whether the commercial version is actually worth the price.

As the source code for both Peach and Sulley are distributed freely with licenses allowing editing and even redistribution following certain constraints, their functionality could be extended with further development. As of now, one of the major limiting factors in Sulley is the missing support for for transport layer injectors, which could be developed and added into the code. Also, both Peach and Sulley are missing a graphical user interface which hinders the user experience. A simple GUI for both could be developed.

Further instrumentation methods are also something all of the tested frameworks would benefit from. The process-based instrumentation method that was developed for this thesis to use with Traffic Capture Fuzzer could be extended further to include support for running the code in a debugger, support for more operating systems, and more. In contrast, Peach and Sulley, which already have process-based instrumentation methods, could further benefit from having a valid-case instrumentation method such as the one used in TCF.

In addition to the metrics used in this thesis, in future a method could be developed to measure protocol coverage in addition to measuring basic code coverage. Measuring protocol coverage would require a monitorable test target that itself covers the protocol fully and is able to monitor the parts which the fuzzers are able to access. This sort of metric would give a clear understanding of how well the fuzzer manages to cover all of the different parts of the protocol, including all of the messages and message sequences.

8. CONCLUSION

The purpose of this thesis work was to analyze fuzzing frameworks comparatively. The main focus was placed on effectiveness and deployment in further defined terms.

To achieve this, a collection of fuzzing frameworks were chosen as test subjects. The chosen frameworks included two free frameworks with their source code available, Peach and Sulley, and one commercial solution, Traffic Capture Fuzzer (TCF). The fuzzing frameworks all perform different mixtures between generational and template-based fuzzing. The frameworks were used to develop test suites for three different network protocols.

During the test suite development process, the frameworks were observed to be remarkably different concerning the development complexity. The development with Sulley was very straightforward and compact in terms of needed lines of code, whereas Peach required a considerable amount of repetition when defining the template and test case sequences. The same repetitive development cycle was present in TCF, mainly because all of the different test case sequences are in practice unique test suites.

In order to evaluate the results, a set of metrics was defined, including code coverage measurements, test effectiveness in terms of crashes, performance and anomaly quality, overall framework quality and usability, as well as the more subjective development and deployment complexity.

A testing environment was defined and the developed test suites were tested against multiple test targets, and closely instrumented to produce the required results. The results were assessed based on the previously defined metrics.

The obtained test results showed that the developed test suites were able to find crashes and vulnerabilities from the chosen test targets. The commercial TCF proved to be the most reliable on average, being at the top or very near in all the tests. Peach was proven effective as well, though its runtime performance turned out to be poor compared to the others. Sulley's test material was not as effective or extensive and it did not find any reproducible crashes.

9. REFERENCES

[1] Beizer B. (1990) Software Testing Techniques. International Thomson Computer Press, second ed.

[2] Takanen A., DeMott J. & Miller C. (2008) Fuzzing for Software Security Testing and Quality Assurance. Artech House.

[3] The heartbleed bug (accessed 06.05.2014). URL: `https://heartbleed.com/`.

[4] Openssl: The open source toolkit for ssl/tls (accessed 26.02.2014). URL: `https://www.openssl.org/`.

[5] Lyu M.R. (2007) Software reliability engineering: A roadmap. In: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, Washington, DC, USA, pp. 153–170. URL: `http://dx.doi.org/10.1109/FOSE.2007.24`.

[6] Tretmans G.J. (1992) A formal approach to conformance testing .

[7] Avritzer A., Kondek J., Liu D. & Weyuker E.J. (2002) Software performance testing based on workload characterization. In: Proceedings of the 3rd international workshop on Software and performance, ACM, pp. 17–24.

[8] Weyuker E. & Vokolos F. (2000) Experience with performance testing of software systems: issues, an approach, and case study. Software Engineering, IEEE Transactions on 26, pp. 1147–1156.

[9] Lei B., Li X., Liu Z., Morisset C. & Stolz V. (2010) Robustness testing for software components. Science of Computer Programming 75, pp. 879–897.

[10] Mukherjee A. & Siewiorek D.P. (1997) Measuring software dependability by robustness benchmarking. Software Engineering, IEEE Transactions on 23, pp. 366–378.

[11] Leung H.K.N. & White L. (1989) Insights into regression testing [software testing]. In: Software Maintenance, 1989., Proceedings., Conference on, pp. 60–69.

[12] Valgrind (accessed 27.02.2014). URL: `http://valgrind.org/`.

[13] Godefroid P., Levin M.Y. & Molnar D. (2012) Sage: Whitebox fuzzing for security testing. Queue 10, pp. 20:20–20:27. URL: `http://doi.acm.org/10.1145/2090147.2094081`.

[14] Bekrar S., Bekrar C., Groz R. & Mounier L. (2011) Finding software vulnerabilities by smart fuzzing. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pp. 427–430.

[15] Godefroid P., Kiezun A. & Levin M.Y. (2008) Grammar-based whitebox fuzzing. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, ACM, New York, NY, USA, pp. 206–215. URL: `http://doi.acm.org/10.1145/1375581.1375607`.

[16] Godefroid P. (2007) 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, ACM, New York, NY, USA, pp. 1–1. URL: `http://doi.acm.org/10.1145/1292414.1292416`.

[17] Nessus vulnerability scanner (accessed 26.02.2014). URL: `http://www.tenable.com/products/nessus`.

[18] Knudsen J. & Varpiola M., Fuzz testing maturity model (accessed 10.01.2014). URL: `http://www.codenomicon.com/resources/ftmm.shtml`.

[19] Provos N., Friedl M. & Honeyman P. (2003) Preventing privilege escalation. In: Proceedings of the 12th USENIX Security Symposium, vol. 12, Washington DC, USA, vol. 12, pp. 231–242.

[20] Provos N. (2003) Improving host security with system call policies. In: Proceedings of the 12th USENIX Security Symposium, vol. 1, Washington, DC, vol. 1, p. 10.

[21] Rossow C. (2014) Amplification hell: Revisiting network protocols for ddos abuse. In: NDSS Symposium 2014.

[22] Juuso A.M. & Takanen A. (2010), Building secure software using fuzzing and static code analysis (accessed 10.01.2014). URL: `http://www.codenomicon.com/resources/whitepapers/codenomicon-wp-fuzzing-and-static-code-analysis-20100811.pdf`.

[23] Protos - security testing of protocol implementations (accessed 10.01.2014). URL: `https://www.ee.oulu.fi/research/ouspg/Protos`.

[24] Kaksonen R. (2001) A functional method for assessing protocol implementation security. Technical Research Centre of Finland.

[25] Monkey lives (accessed 14.03.2014). URL: `http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt`.

[26] Miller B., Fredriksen L. & So B. (1990) An empirical study of the reliability of unix utilities. Communications of the ACM 33.

[27] Protos test-suite: c06-snmpv1 (accessed 10.01.2014). URL: `https://www.ee.oulu.fi/research/ouspg/PROTOS_Test-Suite_c06-snmpv1`.

[28] Protos remote snmp attack tool (accessed 10.01.2014). URL: `http://www.iss.net/threats/advise110.html`.

[29] Cisco ios snmp protos snmpv1 test suite trap handling vulnerability detected (accessed 10.01.2014). URL: `http://www.iss.net/security_center/reference/vuln/cisco-snmp-improper-trap-handling.htm`.

[30] Jiang G. (2002) Multiple vulnerabilities in snmp. Computer 35, pp. 2–4.

[31] Whelan E. (2002), Snmp and potential asn.1 vulnerabilities (accessed 10.01.2014). URL: `http://www.sans.org/reading-room/whitepapers/protocols/snmp-and-potential-asn-1-vulnerabilities-912`.

[32] Immunity: Knowing you're secure (accessed 25.11.2013). URL: `https://www.immunitysec.com/resources-freesoftware.shtml`.

[33] Gnucitizen general purpose fuzzer (accessed 25.11.2013). URL: `http://www.gnucitizen.org/blog/general-purpose-fuzzer_py/`.

[34] Peach fuzzer homepage (accessed 27.02.2014). URL: `http://peachfuzzer.com/`.

[35] Openrce/sulley (accessed 25.11.2013). URL: `https://github.com/OpenRCE/sulley`.

[36] Defensics. defend. then deploy. | codenomicon defensics (accessed 25.11.2013). URL: `http://www.codenomicon.com/`.

[37] Spirent (accessed 25.11.2013). URL: `http://www.spirent.com/`.

[38] bestorm software security testing tool (accessed 25.11.2013). URL: `http://www.beyondsecurity.com/bestorm.html`.

[39] Lipner S. (2004) The trustworthy computing security development lifecycle. In: Computer Security Applications Conference, 2004. 20th Annual, pp. 2–13.

[40] Lipner S. (2010) Security development lifecycle. Datenschutz und Datensicherheit-DuD 34, pp. 135–137.

[41] Microsoft development lifecycle: Verification (accessed 10.01.2014). URL: `http://www.microsoft.com/security/sdl/process/verification.aspx`.

[42] Howard M. & Lipner S. (2009) The security development lifecycle. O'Reilly Media, Incorporated.

[43] Cisco secure development lifecycle (accessed 10.01.2014). URL: `http://www.cisco.com/web/about/security/cspo/csdl/process.html`.

[44] Vuontisjärvi M., Takanen A., Häyrynen A. & Hytönen K. (2012), Fuzzing as a part of agile software development project (accessed 10.01.2014). URL: `http://www.codenomicon.com/resources/whitepapers/2012-agile-fuzzing.shtml`.

[45] Tassey G. (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project 7007.

[46] Kellet A. (2013), The cost of zero-day attacks in the financial sector (accessed 10.01.2014). URL: `http://www.codenomicon.com/resources/whitepapers/2013-ovum-whitepaper-zerodays-in-finance.pdf`.

[47] Howard M. (2004) Building more secure software with improved development processes. Security Privacy, IEEE 2, pp. 63–65.

[48] Smith C. & Francia III G. (2012) Security fuzzing toolset. In: Proceedings of the 50th Annual Southeast Regional Conference, ACM-SE '12, ACM, New York, NY, USA, pp. 329–330. URL: `http://doi.acm.org/10.1145/2184512.2184589`.

[49] Wang T., Wei T., Gu G. & Zou W. (2011) Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. ACM Trans. Inf. Syst. Secur. 14, pp. 15:1–15:28. URL: `http://doi.acm.org/10.1145/2019599.2019600`.

[50] Knudsen J. (2013) Make software better with fuzzing. ISSA Journal , pp. 28–33URL: `http://www.codenomicon.com/news/editorial/Make%20Software%20Better%20with%20Fuzzing.pdf`.

[51] Ciupa I., Leitner A., Oriol M. & Meyer B. (2007) Experimental assessment of random testing for object-oriented software. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, ACM, New York, NY, USA, pp. 84–94. URL: `http://doi.acm.org/10.1145/1273463.1273476`.

[52] Juuso A.M. & Varpiola M., Fuzzing best practices: Combining generation and mutation-based fuzzing (accessed 10.01.2014). URL: `http://www.codenomicon.com/resources/whitepapers/generation-and-mutation-based-fuzzing.shtml`.

[53] Dod anti-tamper (accessed 25.11.2013). URL: `https://at.dod.mil/`.

[54] Whitehouse O., Introduction to anti-fuzzing: A defence in depth aid (accessed 20.01.2014). URL: `https://www.nccgroup.com/en/blog/2014/01/introduction-to-anti-fuzzing-a-defence-in-depth-aid/`.

[55] Ormandy T., Making software dumber (accessed 20.01.2014). URL: `http://taviso.decsystem.org/making_software_dumber.pdf`.

[56] Lämsä J., Kaksonen R. & Kortti H., Codenomicon robustness testing - handling the infinite input space while breaking software (accessed 10.01.2014). URL: `http://www.codenomicon.com/resources/whitepapers/codenomicon-wp-fuzzing-and-static-code-analysis-20100811.pdf`.

[57] Case J.D., Fedor M., Schoffstall M.L. & Davin J.R. (1990), Rfc1157: A simple network management protocol (snmp) (accessed 10.01.2014). URL: `http://www.ietf.org/rfc/rfc1157.txt`.

[58] Common vulnerability scoring system (cvss-sig) (accessed 25.02.2014). URL: `http://www.first.org/cvss`.

[59] The mit license (mit) (accessed 26.02.2014). URL: `http://opensource.org/licenses/MIT`.

[60] .net (accessed 25.11.2013). URL: `http://www.microsoft.com/net`.

[61] Mono (accessed 27.02.2014). URL: `http://www.mono-project.com/`.

[62] Peach fuzzer(tm) overview (accessed 27.02.2014). URL: `http://peachfuzzer.com/pdf/Peach-Overview-DejaVuSecurity-Datasheet-2014.pdf`.

[63] Peach fuzzer mutators (accessed 27.02.2014). URL: `http://old.peachfuzzer.com/v3/Mutators.html`.

[64] Postel J. & Reynolds J., Rfc2960: Stream control transmission protocol (sctp) (accessed 27.02.2014). URL: `http://www.ietf.org/rfc/rfc2960.txt`.

[65] Arkko J. & Loughney J., Rfc6733: Diameter base protocol (accessed 27.02.2014). URL: `http://tools.ietf.org/html/rfc6733`.

[66] Ylonen T., Rfc4251: The secure shell (ssh) protocol architecture (accessed 27.02.2014). URL: `http://www.ietf.org/rfc/rfc4251`.

[67] Gdb - the gnu project debugger (accessed 10.1.2014). URL: `http://www.sourceware.org/gdb/`.

[68] Vmware (accessed 25.11.2013). URL: `http://www.vmware.com/`.

[69] Welcome to python.org (accessed 27.02.2014). URL: `https://www.python.org/`.

[70] Amini P. & Portnoy A. (2007), Fuzzing sucks! introducing sulley fuzzing framework (accessed 25.11.2013). URL: `http://fuzzinginfo.files.wordpress.com/2012/05/introducing_sulley.pdf`.

[71] MARIANI B. (2013), Fuzzing: An introduction to sulley framework (accessed 25.11.2013). URL: `https://www.htbridge.com/publication/The-Sulley-Fuzzing-Framework.pdf`.

[72] Freizer A., Karlton P. & Kocher P., Rfc6101: The secure sockets layer (ssl) (accessed 25.11.2013). URL: `http://tools.ietf.org/html/rfc6101`.

[73] Rey E. & Mende D., Advanced protocol fuzzing - what we learned when bringing layer2 logic to spike land (accessed 25.11.2013). URL: `https://www.ernw.de/download/l2_fuzzing_shmoo08.pdf`.

[74] Ernw - providing security. (accessed 25.11.2013). URL: https://www.ernw.de/.

[75] Netzob (accessed 27.02.2014). URL: http://www.netzob.org/.

[76] Traffic capture fuzzing with software development kit (sdk) (accessed 26.11.2013). URL: http://www.codenomicon.com/defensics/traffic-capture-with-sdk.shtml.

[77] Wireshark (accessed 25.11.2013). URL: http://www.wireshark.org/.

[78] The traffic capture test suite with sdk data sheet (accessed 26.11.2013). URL: http://www.codenomicon.com/products/capture.shtml.

[79] Postel J. & Reynolds J., Rfc959: File transfer protocol (ftp) (accessed 25.11.2013). URL: http://www.ietf.org/rfc/rfc959.txt.

[80] Rosenberg J., Schulzrinne H., Camarillo G., Johnston A., Peterson J., Handley R.S.M. & Schooler E., Rfc3261: Sip: Session initiation protocol (accessed 25.11.2013). URL: http://www.ietf.org/rfc/rfc3261.txt.

[81] Locator/id separation protocol (lisp) - documents (accessed 25.11.2013). URL: https://datatracker.ietf.org/wg/lisp/.

[82] Bhushan A., Rfc114: A file transfer protocol (accessed 25.11.2013). URL: http://tools.ietf.org/html/rfc114.

[83] Postel J., Rfc765: File transfer protocol (accessed 25.11.2013). URL: http://tools.ietf.org/html/rfc765.

[84] Ford-Hutchinson P., Rfc4217: Securing ftp with tls (accessed 25.11.2013). URL: http://tools.ietf.org/html/rfc4217.

[85] Dierks T. & Rescorla E., Rfc5246: The transport layer security (tls) protocol version 1.2 (accessed 25.11.2013). URL: http://tools.ietf.org/html/rfc5246.

[86] Handley M., Jacobson V. & Perkins C., Rfc4566: Sdp: Session description protocol (accessed 25.11.2013). URL: http://tools.ietf.org/html/rfc4566.html.

[87] Rfc3550: Rtp: A transport protocol for real-time applications (accessed 25.11.2013). URL: http://www.ietf.org/rfc/rfc3550.txt.

[88] Iannone L., Saucez D. & Bonaventure O. (2011) Implementing the locator/id separation protocol: Design and experience. Computer Networks 55, pp. 948–958.

[89] Farinacci D., Fuller V., Meyer D. & Lewis D., Rfc6830: The locator/id separation protocol (lisp) (accessed 25.11.2013). URL: http://tools.ietf.org/html/rfc6830.

[90] Soto I., Bernardos C.J. & Calderón M. (2007) Ipv6 network mobility. The Internet Protocol Journal (IPJ) 10, pp. 16–27.

[91] Gundavelli S., Leung K., Devarapalli V., Chowdhury K. & Patil B., Rfc5213: Proxy mobile ipv6 (accessed 18.03.2014). URL: `http://tools.ietf.org/html/rfc5213`.

[92] Cisco location/id separation protocol early deployment software releases (accessed 19.03.2014). URL: `http://www.cisco.com/c/dam/en/us/products/collateral/ios-nx-os-software/ip-routing/prod_bulletin_c25-598556.pdf`.

[93] Locator/id separation protocol architecture (accessed 18.03.2014). URL: `http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/locator-id-separation-protocol-lisp/white_paper_c11-652502.html`.

[94] Enterprise ipv6 transition strategy (accessed 18.03.2014). URL: `http://www.cisco.com/c/dam/en/us/products/collateral/ios-nx-os-software/locator-id-separation-protocol-lisp/white_paper_c11-629044.pdf`.

[95] Ipv6 transition and coexistence using lisp (accessed 18.03.2014). URL: `http://www.cisco.com/c/dam/en/us/products/collateral/ios-nx-os-software/locator-id-separation-protocol-lisp/whitepaper_c11-665752.pdf`.

[96] Devadas S., Ghosh A. & Keutzer K. (1997) An observability-based code coverage metric for functional simulation. In: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, IEEE Computer Society, pp. 418–425.

[97] Tikir M.M. & Hollingsworth J.K. (2002) Efficient instrumentation for code coverage testing. In: ACM SIGSOFT Software Engineering Notes, vol. 27, ACM, vol. 27, pp. 86–96.

[98] Chen M.H., Lyu M.R. & Wong W.E. (2001) Effect of code coverage on software reliability measurement. Reliability, IEEE Transactions on 50, pp. 165–170.

[99] Gcc, the gnu compiler collection - gnu project - free software foundation (fsf) (accessed 10.1.2014). URL: `http://gcc.gnu.org/`.

[100] C test coverage tool (accessed 10.01.2014). URL: `http://www.semanticdesigns.com/Products/TestCoverage/CTestCoverage.html`.

[101] Bullseyecoverage code coverage analyzer (accessed 10.01.2013). URL: `http://www.bullseye.com/`.

[102] Linux test project - coverage: lcov (accessed 10.1.2014). URL: `http://ltp.sourceforge.net/coverage/lcov.php`.

[103] Levenshtein V.I. (1966) Binary codes capable of correcting deletions, insertions and reversals. In: Soviet physics doklady, vol. 10, vol. 10, p. 707.

[104] Damerau F.J. (1964) A technique for computer detection and correction of spelling errors. Commun. ACM 7, pp. 171–176. URL: `http://doi.acm.org/10.1145/363958.363994`.

[105] Bard G.V. (2007) Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In: Proceedings of the Fifth Australasian Symposium on ACSW Frontiers - Volume 68, ACSW '07, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 117–124. URL: `http://dl.acm.org/citation.cfm?id=1274531.1274545`.

[106] Oracle vm virtualbox (accessed 25.11.2013). URL: `https://www.virtualbox.org/`.

[107] The world's most popular free os (accessed 25.11.2013). URL: `http://www.ubuntu.com/`.

[108] Pure-ftpd (accessed 25.11.2013). URL: `http://www.pureftpd.org/`.

[109] Proftpd (accessed 25.11.2013). URL: `http://www.proftpd.org/`.

[110] Lispmob (accessed 25.02.2014). URL: `http://lispmob.org/`.

[111] Cabellos A., Natal A.R., Jakab L., Ermagan V., Natarajan P. & Maino F., Lispmob: Mobile networking through lisp (accessed 25.02.2014). URL: `http://lispmob.org/sites/default/files/users/user1/documents/LISPmob_Whitepaper.pdf`.

[112] Asterisk.org (accessed 27.02.2014). URL: `http://www.asterisk.org/`.

[113] Sj labs, the softphone factory (accessed 27.02.2014). URL: `http://www.sjphone.org/`.

[114] Epydoc (accessed 12.11.2013). URL: `http://epydoc.sourceforge.net/`.