Tampere University

Nico Ahola

AUTOMATION OF DEFENSICS FOR PROTOCOL IMPLEMENTATION FUZZING

Master's Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Marko Helenius
Bilhanan Silverajan
April 2022

ABSTRACT

Nico Ahola: Automation of Defensics for protocol implementation fuzzing
Master's Thesis
Tampere University
Master's Degree Programme in Information Technology
April 2022

Fuzzing is a security testing method that has existed for decades and that has been widely adopted by the industry. Its goal is to expose vulnerabilities by generating inputs that cause unexpected behaviour in a system, e.g. software crashes. There exists several different fuzzing types, one of which is network protocol fuzzing. A network protocol fuzzer tries to find flaws in protocol implementations. Defensics is one such fuzzer.

A Nokia team had been using Defensics manually via GUI as part of their product's security testing. Operating the GUI had taken notable time. The GUI also has limitations not present when using Defensics via its CLI or HTTP API. Another challenge with Defensics had been its execution time. Especially when Defensics marked test cases as skipped or failed seemed to cause extremely slow behaviour.

To address the challenges, Defensics was added into existing CI process and its suites were configured in a way that speeds up fuzzing process when many cases are marked as skipped or failed. As part of the work, a Robot test suite and a Python program were created. The CI pipeline that executes Defensics calls the Robot test suite which then calls the program. The program can execute Defensics processes in parallel using Defensics CLI and multithreading.

A comparison was done between old and new suite configurations. The results show that execution time has slightly improved when many skipped cases are encountered. Even with the improvement, fuzzing was concluded to be too slow to be fully executed for every product release candidate. Therefore, two pipelines exist: one for executing a small subset of cases for release candidates and another for full execution on a weekly or on-demand basis.

Keywords: defensics, fuzzing, automation, security

TIIVISTELMÄ

Nico Ahola: Defensicsin automatisointi protokollatoteutuksien fuzz-testaukseen
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Huhtikuu 2022

Fuzz-testaus on vuosikymmeniä vanha ja laajasti käytössä oleva tietoturvatestauksen metodi. Sen tavoite on paljastaa haavoittuvuuksia luomalla syötteitä, jotka aiheuttavat odottamattomia reaktioita kohdejärjestelmässä. Odottamaton reaktio voi olla esimerkiksi sovelluksen kaatuminen. On olemassa monia eri fuzz-testauksen alalajeja ja näistä yksi on protokollien fuzz-testaus. Tällaisessa fuzz-testauksessa tarkoitus on löytää vikoja protokollatoteutuksista. Yksi tämän tyyppiseen fuzz-testaukseen erikoistunut työkalu on Defensics.

Eräs Nokian työryhmä oli käyttänyt Defensicsiä manuaalisesti sen käyttöliittymän kautta osana heidän tuotteensa tietoturvatestausta. Käyttöliittymän käyttö oli vaatinut huomattavasti aikaa. Käyttöliittymässä on myös rajoituksia, joita ei ole Defensicsin komentorivikäyttöliittymässä. Toinen haaste Defensicsin käytössä on ollut fuzz-testauksen hitaus. Erityistä hitautta on esiintynyt Defensicsin merkitessä testitapauksia ohitetuiksi tai epäonnistuneiksi.

Näihin haasteisiin vastattiin liittämällä Defensics osaksi jatkuvaa integrointia (engl. continuous integration, CI) ja Defensicsin asetukset määritettiin siten, että fuzz-testaus etenee nopeammin niissä tilanteissa, joissa moni testitapaus merkataan ohitetuksi tai epäonnistuneeksi. Osana työtä luotiin Robot framework testitiedosto ja Python-ohjelma. Robot-tiedostoa, joka suorittaa Python-ohjelman, kutsutaan CI-putkesta. Python-ohjelma pystyy suorittamaan Defensics-prosesseja rinnakkaisesti Defensicsin komentorivikäyttöliittymän ja monisäikeistyksen avulla.

Työn lopuksi uusia asetuksia vertailtiin vanhoihin asetuksiin. Tuloksista ilmenee, että suoritusaika on hiukan parantunut kun ohitettuja testitapauksia esiintyy paljon. Edes parannuksen kanssa fuzz-testaus ei kuitenkaan ole tarpeeksi nopeaa suoritettavaksi jokaiselle uudelle versiolle tuotteesta. Tämän seurauksena luotiin kaksi CI-putkea: yksi lyhyiden testien ajoon jokaiselle uudelle versiolle tuotteesta ja toinen täysimittaisten testien ajoon viikottain tai aina tarpeen vaatiessa.

Avainsanat: defensics, fuzz-testaus, automaatio, tietoturva

CONTENTS

1. Introduction . . . 1
   - 1.1 Background . . . 1
   - 1.2 Research questions and methodologies . . . 2
   - 1.3 Objectives and outline . . . 2
   - 1.4 Related work . . . 3
2. Security testing background . . . 5
   - 2.1 Software life cycle management . . . 5
   - 2.2 Security testing . . . 8
   - 2.3 Nokia processes . . . 10
   - 2.4 Testing requirements . . . 12
   - 2.5 Testing environment . . . 14
   - 2.6 Automated testing . . . 15
3. Fuzzing . . . 20
   - 3.1 Robustness . . . 20
   - 3.2 Robustness testing . . . 21
   - 3.3 Fuzz testing . . . 23
     - 3.3.1 Basic concepts . . . 23
     - 3.3.2 Workflow . . . 24
     - 3.3.3 White, grey and black box fuzzing . . . 27
     - 3.3.4 Mutation- and generation-based fuzzing . . . 30
     - 3.3.5 Techniques . . . 31
   - 3.4 Protocol fuzzing . . . 34
   - 3.5 Fuzz testing maturity model . . . 36
4. Defensics . . . 39
   - 4.1 Introduction . . . 39
   - 4.2 Installation and licensing . . . 40
   - 4.3 Configuration . . . 41
     - 4.3.1 Pre-execution . . . 41
     - 4.3.2 Post-execution . . . 45
   - 4.4 Current Challenges . . . 46
5. Automating Defensics . . . 48
   - 5.1 Defensics configuration . . . 48
     - 5.1.1 Test suites . . . 48
     - 5.1.2 Test suite configuration . . . 51

5.2 Executing Defensics in CI 53
5.3 Robot test cases 55
5.4 Code implementation 56
6. Analysis and discussion 61
6.1 Execution time comparison 61
6.2 Comparison to manual workflow 63
6.3 Limitations 64
6.4 Future work 65
7. Conclusions 67
References 69

LIST OF FIGURES

2.1 OWASP SDLC phases 6
2.2 Security testing as a part of (S)SDLC 9
2.3 Nokia SSDLC 11
2.4 Bare metal Topology 14
2.5 Cloud Topology 15
2.6 General CI workflow in the given environment 16
2.7 CI Pipelines and CRT 17

3.1 Fuzzing Workflow 25

5.1 Defensics as part of CI 54
5.2 DefensicsPy UML Class Diagram 58

LIST OF TABLES

2.1 Tools 18

3.1 Fuzzing related terminology 24
3.2 Fuzz Testing Maturity Model Levels 37

4.1 Instrumentation methods 43

5.1 Possible test suites 49
5.2 Selected suites 50

6.1 Suite configuration comparison for mixed suites 62
6.2 Suite configuration comparison for TCP suites 63

LISTINGS

2.1 Robot framework example suite 19
5.1 Robot test cases for Defensics Robot test suite 55

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
CD	Continuous Delivery/Development
CFA	Control Flow Analysis
CFG	Control Flow Graph
CGF	Coverage-based Grey box Fuzzing
CI	Continuous Integration
CIA (triad)	Confidentiality, Integrity and Availability
CLI	Command Line Interface
CRASH (criteria)	Catastrophic, Restart, Abort, Silent, Hindering
CRT	Continuous Regression Testing
CVSS	Common Vulnerability Scoring System
DAST	Dynamic Analysis Security Testing
DoS	Denial of Service
DSE	Dynamic Symbolic Execution
FCS	Fuzz Configuration Scheduling
FTMM	Fuzz Testing Maturity Model
GUI	Graphical User Interface
MBST	Model-Based Security Testing
MBT	Model-Based Testing
MSDL	Microsoft Security Development Lifecycle
OID	Object Identifier
OWASP	Open Web Application Security Project
PLM	Product Life cycle Management
PUT	Program Under Test
RFC	Request for Comments
SAST	Static Analysis Security Testing
SBF	Stateful Black box Fuzzing

SCGF	Stateful Coverage-based Grey box Fuzzing
SDLC	Software Development Life Cycle
SSDLC	Secure Software Development Life Cycle
SSP	Seed Selection Problem
SUT	System Under Test
SVM	Security Vulnerability Monitoring
VM	Virtual Machine

1. INTRODUCTION

The first chapter introduces the topic of the thesis and other relevant information. First, general background about the topic is given. Secondly, existing challenges are introduced and research questions are formed. Research methodologies are also described. Third, objectives of the thesis and thesis outline are set out. Finally, other studies related to this thesis are briefly discussed.

1.1 Background

Fuzzing is a security testing technique that has been present for over three decades. It is widely used in the industry and research on fuzzing is only increasing [1]. Objective of fuzzing is to generate data that will cause unexpected behaviour in the target system, such as a crash. In its simplest form fuzzing is nothing but generating random data and inputting it into a program. More complex techniques produce high quality inputs that have a higher chance of causing a crash. The inputs may even evolve based on the target system's reactions. Crashes are the most obvious reactions, but they can be more subtle too. The more subtle the reactions, the more target analysis must be performed by the fuzzer. However, sometimes observing subtle reactions is not possible. For example, when fuzzing network protocol implementations of a separate target machine it is possible that only inputs and outputs can be observed. This type of fuzzing is called black box fuzzing.

This thesis is done for a team of a Nokia organization, that develops a software product for mobile networks. The product is installed on a host machine running a certain Linux distribution. Software testing, including security testing, has been part of the product's life cycle for a while, but the testing has been mostly manual. Automation of the testing has started only recently by using continuous integration (CI) practices. As one part of the product's security testing the team has conducted black box protocol fuzzing using Defensics fuzzer, which is one of the tools not part of automated testing yet. But now there is a need to automate Defensics too. The automation will be done as purely individual work.

1.2 Research questions and methodologies

For now, Defensics has been used via its GUI application. Operating the GUI is time-consuming and embodies limitations not present in alternative execution methods, i.e., execution via Defensics command line interface (CLI) or HTTP API. Since these two alternative methods exist, automation will be done using one of them. Which one and how exactly should it be used is an open question. Another question that has risen during manual execution is whether the execution time can be shortened. Currently executing Defensics is by far the slowest testing method and therefore has a risk of extensively prolonging automated execution. To formalize these questions into research questions of this thesis:

- How should Defensics protocol fuzzing be integrated into existing CI process?
- Is it possible to speed up fuzzing performed by Defensics? If so, how?

Two primary research methodologies were used in this thesis: literature review and practical work. A large of amount of literature was reviewed for gathering information about security testing, software testing in general and fuzz testing. Part of this literature was used as source material, mainly for the first two chapters. Tampere University's Andor and Google Scholar were used as the main search engines. Search terms were focused on security testing, testing automation, robustness and fuzz testing. The approach used was however not systematic and the precise terms used are not available. Defensics documentation was used as source when writing about Defensics. This documentation is not publicly available since it requires access to Defensics. Most of the practical work was writing Python code, but a significant amount of time went also into creating test cases with Robot framework and configuring the pipelines in Gitlab. The testing environment and a security pipeline already existed before, but all work regarding Defensics was done as individual work for this thesis.

1.3 Objectives and outline

The main objective of this thesis is to create a program for running Defensics commands. This program is then integrated as part of the existing automated testing workflow. The other objective is to create the program and configure Defensics in such a manner that execution time is significantly shorter than currently. Ideally Defensics could be run multiple times a day without reducing test coverage. Defensics allows fuzzing of file formats, but for this thesis only protocol fuzzing of Defensics is analyzed. The theory part of the thesis will, however, deal with other types of fuzzing too. The product will be treated as black box, i.e., trying to infer control flow or such is out of the scope of this thesis. However, monitoring the host machine is not out of the question.

The structure of the thesis is as follows: first, chapter 2 briefly discusses theory behind security testing and its automation. The chapter also summarizes requirements and environment set for the work. Chapter 3 then moves onto theory behind fuzz testing. The chapter moves from general robustness testing onto fuzz testing and its different types and techniques. As the last part of the chapter, fuzz testing maturity model is introduced. Chapter 4 is the last theory chapter as it discusses Defensics and its features and requirements. The fifth chapter describes what was actually done in this thesis: Defensics configurations, automatic execution workflow of Defensics and the written code. Chapter 6 is for analysis and general discussion of the work done. Finally, everything is concluded in chapter 7.

1.4 Related work

A work closely related to this thesis was conducted by Sorsa [2]. In her work Sorsa designed and implemented a fuzzing framework into an existing CI environment. Fuzzer used in the work was Radamsa, an open source general purpose fuzzer [3]. Similar to this thesis, Robot Framework was used as testing automation framework. The used CI server was, however, Jenkins. Besides differences in used technologies, no fuzzing had been conducted prior the work. Hence, new implementation errors were found from the fuzzed product, although detailed analysis was out of the scope of the work. Another important difference compared to this work is that the fuzzed protocols were proprietary. For future work Sorsa suggests Defensics SDK as an alternative for Radamsa. Defensics SDK makes it possible to fuzz proprietary protocols by creating custom test suites. Because no proprietary protocols are fuzzed in this work, Defensics SDK is not discussed further.

No other works related to integrating a fuzzer into a CI environment were identified using the described research methodologies. However, other works related to Defensics and its automation exist. Oka et al. [4] created a prototype that automates the process of mapping security testing requirements and the corresponding test case(s) . The prototype uses Jenkins to read test case requirements and launch appropriate Defensics test suite. The requirements are read from an application lifecycle management (ALM) tool and results are fed back into it after fuzz testing. The prototype uses preconfigured Defensics testplans, instead of dynamically creating parameters for test suites. This may require some manual work later if changes to testplans are necessary. Rossi [5] created scripts as part of his thesis for automating certain Defensics related tasks that the target company had done manually. Execution of Defensics itself however was not automated, i.e., fuzz testing was not part of any CI environment.

There are also many works where Defensics is used but the automation perspective is not considered. In such works the focus is commonly not in Defensics but in some other system. Such systems include electronic control units used in automotive industry [6],

various components used in smart grids [7] and programmable logic controllers used in nuclear power plants [8]. In the mentioned works Defensics is used as comparison to another fuzzer or for exposing vulnerabilities.

When it comes to fuzz testing in general, there is a large amount of studies with several different focus areas. Developing a new fuzzer (e.g. [9]), writing more efficient fuzzing algorithms (e.g. [10]) and comparing existing fuzzers (e.g. [11]) are examples of different focus areas. Due to the vast amount of studies, literature reviews and surveys help to understand the big picture. Manes et al. [1], Liang et al. [12] and Li et al. [13] have conducted such papers, to name a few. One of the most recent literature reviews concerns machine learning based fuzzing [14]. Regarding this thesis, one of the most interesting areas is network protocol fuzzing. Munea et al. [15] reviewed studies on network protocol fuzzing and compared four fuzzers based on several criteria. Based on the comparison and used criteria, an ideal network protocol fuzzing technique combination was introduced. However, the authors state that creating a fuzzer that matches the ideal combination may not be practical.

2. SECURITY TESTING BACKGROUND

This chapter goes through theory behind security testing. First is discussed how security testing is related to general software life cycle management. Then, the topic moves on to security testing itself and its different methods. Thirdly, previous topics are analyzed from Nokia perspective. After that, the security testing requirements and environments of this work are introduced. Finally, concepts related to testing automation are gone through from the perspective of the given environment.

2.1 Software life cycle management

Security and secure products play a big role in our current society, where the amount of security flaws in products keeps rising [16]. When product security flaws are exploited, the company's brand can be damaged [17]. Positive brand image has been shown to have a direct correlation to customer loyalty, which is also impacted by trust [18]. Therefore it is important to create secure products that gain customer trust and create a trustworthy brand.

To reach the objective of having secure products, there exists processes, policies and guidelines. A high-level process for managing and developing products and information related to them is called product life cycle management (PLM). It is a collection of instructions, subprocesses and guidelines aimed to reach the high-level goals of a business. Besides company goals, PLM also depends on business problems and strategies. Because of this, PLMs between companies can differ vastly. However, the core objective stays the same: handle product life cycle effectively. In practice, this means, for example, improving product quality, minimizing waste and maximizing product value. [19]

One commonly used process for software development is called software development life cycle (SDLC). It is an abstract process that describes phases of software from an idea to maintaining it. There exists several high-level SDLC models, such as waterfall and agile. The former emphasises planning and linear workflow, while the latter prioritises continuous planning and development, among other stages. Businesses usually follow one model during software's life cycle, but implementations differ. After all, every software and its environment is different. [20]

Open Web Application Security Project (OWASP) is a nonprofit foundation that works to improve software security. OWASP Testing guide [21] depicts a general SDLC model consisting of 5 stages. These stages are define, design, develop, deploy and maintain as seen in figure 2.1.

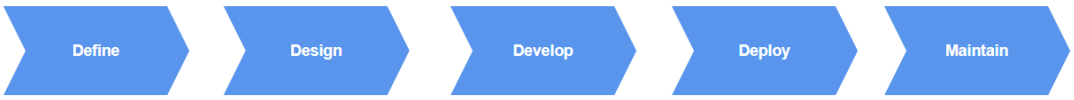

Figure 2.1. *OWASP SDLC phases*

Before any actual stage, company policies, standards and other documents concerning software life cycle should be defined. These high-level documents guide processes during the life cycle and set the objectives. After this identifying current problems and defining requirements for the software can be done. It is also important to evaluate project feasibility to not waste time and resources. Then, during design, software architecture, documentation, different models (e.g. UML models). etc. can be created. Design stage is perhaps the most important stage as design decisions can become difficult and expensive to change during later stages. [21]

The actual implementation of design happens in development stage. The level of detail in previous stages assigns how much decisions are left for developers. Unit and system testing is part of this stage. Application testing happens at deployment stage. Once software has been deployed, it needs to be maintained. Maintaining includes updating the software, performing health checks, doing regression testing etc. [21]

The problem with SDLC from security perspective is that security is usually implemented after development. This is an ineffective approach for securing software [22]. Furthermore, the later the vulnerabilities are found, the more it costs to fix them [23] [24]. Because software gets increasingly complex, it is inevitable that vulnerabilities are found at some point. This problem is addressed by secure software development life cycle (SSDLC), which extends SDLC by incorporating security into software's life cycle.

Incorporating security into SDLC entails several benefits [25]. Bugs and vulnerabilities are found earlier during the life cycle, which implies simpler bug fixes and therefore minimized effort and expenses. The more vulnerabilities are found before release, the less the total amount of vulnerabilities. This makes the software more trustworthy. Trustworthiness and thorough testing can be used as selling points for customers. Additionally, since the total amount of vulnerabilities is minimized, malicious actors have less vulnerabilities to expose. This is important for business's image, which can be greatly damaged if vulnerabilities are exploited and become publicly known [17].

As with SDLC, several different SSDLC models exist. OWASP Testing Guide also has

security actions listed for each of the 5 stages discussed, so the same model is discussed here with the added actions. When implementing a SSDLC, security related policies and standards ought to be included from the beginning. These should be followed same way as other policies and standards during the lifcycle. During definition and design stage all security requirements, designs, models etc. should be created and reviewed from security perspective. Threat and risk assessment should also be made during this stage. [21]

Producing secure code is the main goal of development stage. To achieve this goal, previously defined policies must be followed. In addition, code reviews, static code analysis etc. can help. During deployment penetration testing acts as a final verification method before software release. It is also important to check that no part of application is in debug state anymore. Lastly, when releasing the software and moving into maintenance stage, it must be ensured that the environment change has not affected security. [21]

OWASP Testing Guide provides only one type of (S)SDLC model. Microsoft SDL (MSDL) is another well known SSDLC model created by Microsoft. It used to be a linear model consisting of 7 phases, but nowadays there's 12 different practices in no strict order, instead of clear phases [26]. These practices are:

- Provide Training
- Define Security Requirements
- Define Metrics and Compliance Reporting
- Perform Threat Modeling
- Establish Design Requirements
- Define and Use Cryptography Standards
- Manage the Security Risk of Using Third-Party Components
- Use Approved Tools
- Perform Static Analysis Security Testing (SAST)
- Perform Dynamic Analysis Security Testing (DAST)
- Perform Penetration Testing
- Establish a Standard Incident Response Process

As can be seen, MSDL practices are not as high-level as the stages in OWASP Testing Guide. However, there are a lot of similarities. For example, both models define requirements for policies, metrics, design etc. and they use code analysis and penetration testing for verifying quality. Some MSDL practices are also part of OWASP's SSDLC, although not clearly mentioned. For example, appropriate training and avoiding bad cryptography are mentioned in the Testing Guide, but not explicitly in the SSDLC framework.

Besides the clear structural difference between OWASP framework and MSDL, there are other differences. For instance, MSDL emphasises third-party component security assessment more. Third-party risk assessment is a one whole practice in MSDL, but only briefly mentioned in OWASP Testing Guide. Additionally, OWASP Testing Guide does not mention incident response plan or how to act when a security incident is reported after release. This is the last practice of MSDL and it used to be the last phase in old MSDL model.

As software life cycle progresses from one phase to another, any vulnerabilities are also carried to later phases [27]. If the vulnerabilities are known and risks they possess are accepted, this might not be a problem. However, if the vulnerabilities are unknown, they may possess risks. Often the risks are financial. NIST study concerning economic impacts of poor security testing frameworks from 2002 shows that bugs (vulnerabilites) become more costlier to fix the later they are detected [24]. Main reason being dependencies that are created as software grows. Untying the dependencies at late stages of software life cycle can demand great amount of work.

2.2 Security testing

Although ideally the goal is to remove vulnerabilities before implementation, this is often not feasible, as the growing number of vulnerabilities reported by NIST indicates [28]. Therefore, security testing is a vital part of SSDLC. Security testing aims to verify that previously defined security requirements are met and that security properties work as intended [22]. To thoroughly execute security testing on a software, various techniques should be used to complement each other because there is no single testing technique to cover everything [22]. In addition, testing should happen at different testing stages, e.g. unit, integration and system level.

Bachmann et al. [29] suggest that security testing should be part of all phases of SDLC. However, they continue that during planning and design phases no actual testing happens. Instead, these stages are a prerequisite for the actual security testing executed in later stages. Bachmann et al. suggest various techniques for each of the phases, such as threat modeling, static source code analysis and fuzz testing. Previously discussed MSDL and OWASP Testing Guide follow this ideology as they have requirement definitions, static code analysis and dynamic testing incorporated. OWASP Testing Guide has also regression testing during maintenance, whereas MSDL only proposes readiness for executing incident response plan.

Felderer et al. [22] suggest a similar structure, where testing is part of all phases of SSDLC. Unlike Bachmann et al. though, they add model-based security testing as part of design phase. Figure 2.2 visualizes how different testing types fit into SSDLC stages defined in [21]. The figure has been adapted from [22].

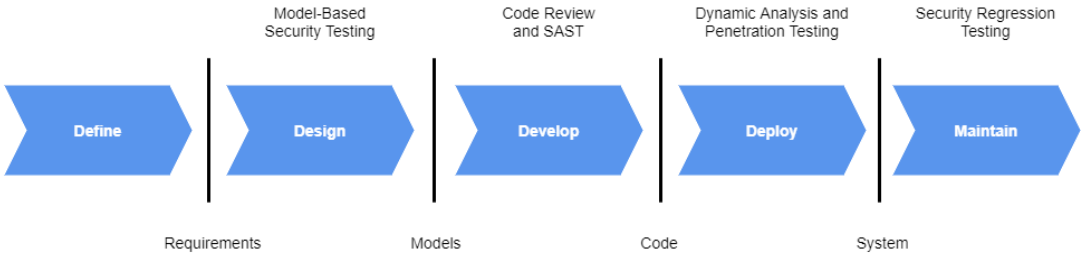

Figure 2.2. *Security testing as a part of (S)SDLC*

During definition and designing, model-based testing (MBT) can be applied. In MBT a test model is created based on the system under test (SUT). The model is then evaluated to see if previously defined requirements are met. This happens by deriving test cases from the model. For instance, UML (or a subset of UML) model can be created from an application, which is then given as an input for a program that generates test objectives and test cases against these objectives [30]. Any design flaws detected during testing can be fixed, which is cheaper than fixing the flaws in later SDLC stages, as discussed earlier.

Model-based security testing (MBST) is like MBT, but only focuses on security properties. The goal is to verify that security requirements are not violated. Because MBST (and MBT) uses a model instead of the real system, false positives can occur during automatic testing. It is therefore important to conduct manual review on any found violations. Besides models used in MBT, MBST may also use property, attacker or vulnerability models. Property models describe the properties that shall not be violated, namely with regard to confidentiality, integrity and availability (CIA triad). Vulnerability models are intentional faults, that attackers can exploit. These attackers and their attacks are described in attacker models. [22]

During development mainly two security testing techniques are applied. These techniques are code review and static application security testing (SAST). Code review means manually inspecting code for finding bugs. From security perspective, the most interesting bugs are vulnerabilities. Furthermore, code review should be done based on system's attack surface and threat analysis to narrow the review process. The automated counterpart for code review is SAST, which tries to find vulnerabilities by applying syntactic and semantic checks. SAST is naturally faster than manual code review and it can effectively cover full source code, but it can also result in false positives and all vulnerabilities might still not be found. Another drawback is that SAST is incapable of finding business logic errors. Hence, one should not blindly trust SAST results. [22]

During deployment security testing consists of dynamic analysis and penetration testing. When conducting penetration testing, tester acts as an attacker. If tester manages to

exploit vulnerabilities, they report them for fixing and attack surface updating. Dynamic analysis includes vulnerability scanning, dynamic taint analysis and fuzz testing. Vulnerability scanning is an automated way of detecting vulnerabilities. For example, a scanner may have a database of vulnerable libraries that it searches in the system. Dynamic taint analysis means marking, or *tainting*, some data and observing if it's used in an insecure context. For example, if user inputted string data is used in SQL query, it might indicate a possible SQL injection vulnerability. Finally, fuzz testing is used for testing system's robustness by injecting invalid or semivalid inputs. The goal is usually to crash the system. [22]

During maintenance, security regression testing is used for verifying security as the system or its environment changes. Three different techniques exist: test suite minimization, test case prioritization and test case selection [22]. The first two are concerned in minimizing test case amount and optimizing test case execution order, respectively. The most popular of these techniques, test case selection, handles selecting a subset of test cases for testing modified parts of the system. Selecting only test cases that are affected by system changes is important, because regression testing is one of the costliest parts of SSDLC [31]. Both test case minimization and selection try to optimize test case amount, but test case selection is only focused on finding cases related to system modifications [32].

From the perspective of this thesis, the most interesting part of security testing is fuzz testing or more generally robustness testing. Both robustness and fuzz testing are closely related, but fuzz testing is actually a subset of robustness testing. This is because robustness testing also includes other methods, which might not even be strictly security related. The two testing types are interested in observing how a system reacts to invalid inputs and/or stressful conditions. The goal is to crash the system or cause other unexpected behaviour. Hence, both robustness and fuzz testing are related to availability part of CIA triad. Fuzz testing is also included in OWASP Testing Guide and Microsoft SDL. Both testing types are discussed more thoroughly in chapter 3.

2.3 Nokia processes

In a large corporation like Nokia, there exists great amount of processes, policies and guidelines. One of these is a process for product, solution and service management. This process describes how products, solutions and services, or generally speaking assets, are defined, developed and managed throughout their life cycle. Because software is one type of asset, this process is analogous to SDLC. Part of this general process is more practical process focused on security, Nokia's implementation of SSDLC. Nokia SSDLC describes how products and systems should be developed to make them secure starting from the beginning of their life cycle. It consists of different stages, where each stage

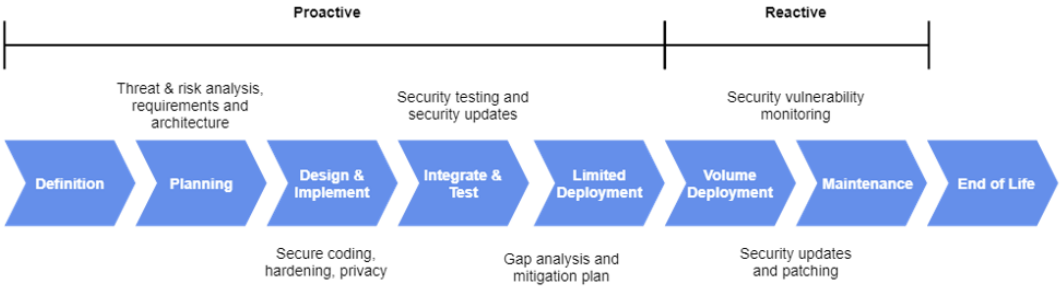

Figure 2.3. *Nokia SSDLC*

contributes to the ultimate goal of creating secure products and systems.

Similar to OWASP testing framework, the Nokia SSDLC process is divided into multiple stages. However, there are 3 additional stages. These stages are initial opportunity definition, limited deployment and the end of life. The change from one stage to another is called a milestone. In other words, milestones are deadlines for each of the stages. Figure 2.3 visualizes Nokia SSDLC, including the security actions at different stages. Security actions are split into proactive and reactive actions.

Proactive actions are taken as precaution, in case something bad happens, e.g. a new vulnerability is discovered in a product. There are 4 proactive security actions, which are:

- Threat & risk analysis, requirements and architecture
- Secure coding, hardening, privacy
- Security testing and security updates
- Gap analysis and mitigation plan

These actions are represented in figure 2.3 close to a SSDLC stage which they concern the most. The actions do not strictly correspond to a single action, however. For example, creation of gap analysis and/or mitigation plan might already start during definition stage.

Nokia SSDLC closely resembles OWASP SSDLC and MSDL. One interesting aspect is privacy, which is part of design and implementation phase of Nokia SSDLC. OWASP SSDLC mentions privacy explicitly only as part of legislative compliance during define and design phases. Also MSDL mentions privacy as part of defining security requirements. All the processes therefore acknowledge privacy, although it does not seem to play a big role. This makes sense though because security and privacy are separate concepts and SSDLC, as the name implies, is focused on security.

Two terms not mentioned in OWASP SSDLC or MSDL are hardening and gap analysis. Hardening is “any process, methodology, product or combination thereof that is used to directly increase the security of existing software” [33]. It can be applied at different levels, for example at source code or operating system level. Gap analysis is a technique

used for comparing the current state against a standard or some predefined requirements [34]. It helps a business to identify any gaps between expected state and actual state. Resources can then be allocated accordingly to close the gaps.

Reactive actions are actions to be taken after something happens. For Nokia's SSDLC these actions are security vulnerability monitoring (SVM) and security updates & patching. SVM collects information on vulnerabilities from public sources to stay constantly updated on latest vulnerabilities. Any found vulnerabilities proceed for a fault management process, where appropriate actions are taken. For some cases this might mean creating a ticket for patching, but in some cases the vulnerability might simply be tagged as false positive. If there is something to fix, there will eventually be a security updates and/or patch. Security updates and patches will be delivered until a product reaches its end of life.

Important notice regarding Nokia SSDLC is that it is not a static model that keeps its structure from one year to another. Instead, it is under constant development. The specific tools and criteria can change whenever they are deemed outdated. For example, if a better alternative is found for performing vulnerability scanning, the old scanner tool will be replaced. Modifications to higher-level structure of the model are of course more rare, but possible if needed.

Security testing conducted during integration & testing stage has several different testing techniques. The techniques that are the most interesting regarding this thesis are robustness and denial of service (DoS) testing. General robustness testing includes fuzz testing, but also other not strictly security related actions. For example, robustness tests can be related to usability of software, which is not a security concern but plays a big role for the end user. Regarding security testing though only fuzz testing is part of robustness testing. DoS testing relates to fuzz testing, as the objective is to find states leading to a crash. The same tool, Defensics, is used for both fuzz and DoS testing. DoS testing specifically, however, will not be discussed further in this thesis.

2.4 Testing requirements

For each security testing type there is a company recommended tool or multiple tools. Although they are recommendations, in practice these are the only tools that should be used. Using the recommended tools also makes things such as licensing and help support easier. Company recommended tool for robustness testing is Defensics, which is a fuzz tester specialized in network protocols. Defensics is also the recommended tool for testing denial of service attack scenarios.

There are also other upper-level requirements that need to be met during security testing. The requirements mainly concern accountability and transparency, i.e. there must exist

proof that testing has been conducted and all the results need to be visible. In practice this implies sufficient documentation. For example, each testing tool should provide some output which states the test results. Furthermore, the outputs should be comparable so that it is possible to say how software quality has progressed.

Product-level requirements are more interested in tool specific matters. For example, scan coverage and scan frequency are such matters. All testing tools offer several configuration possibilities which make it possible to modify what is scanned and what is not scanned, i.e. scan coverage. Ideally every possible interface should be scanned, but in practice time becomes a limiting factor. Hence, scan coverage should cover only the necessary interfaces. Scan frequency is another aspect limited by time constraints. For example, it is usually not possible to scan everything daily as even a single test run might take over 24 hours. However system level tests should be frequent, because they have been shown to reduce software development time especially when previous experience on the software is low [35].

For Defensics, there are no requirements for scan frequency as long as all necessary interfaces are tested. Ideally interfaces are tested once with successful scan results. In practice though there might be scan failures or product updates, which require manual analyzing and rerunning certain interfaces (or all of them). Every update practically creates a new product release candidate that must go through security testing. The goal is to release a candidate that is as secure as possible. Even thorough scanning does not provide perfect security, but it does increase confidence on the security and robustness of the product.

The only requirements regarding Defensics are related to testing period, analyzing failed cases and documentation. First, necessary interfaces must be tested within a system testing period, which varies but usually lasts a couple of weeks. The interfaces are determined using port scanning and product documentation. All open ports are an attack vector, which an attacker could exploit. Therefore it is important to be aware of all the possible attack vectors and test them as much as possible. Secondly, all failures must go through manual analyzing to determine if they are legit or false positive. Third and final requirement comes from upper-level: there must exist sufficient documentation from all testing. This includes documenting both passed and failed cases, and possible reruns.

When it comes to automated testing and Defensics, testing requirements are related to execution time. There are no clear limits, but it should be expected that testing is done at least daily. Some Defensics scans have exceeded 24 hours, which is why there must be an option to limit execution time. The product that is tested also has many interfaces, which implies that parallel execution must be supported.

2.5 Testing environment

There exists two different environments for security testing: bare metal and cloud environment. Bare metal environment has physical machines while cloud environment has virtual machines. Both environments are purely for security testing. They are isolated into their own subnets to not cause interference to other environments. This is important because interference can happen both ways. For example, security test results could be flawed due to extra ports opened by other tests or an important server could crash due to fuzz testing. Even if there is no crash, security testing, namely fuzz testing, might leave servers in an unexpected state [36].

The environments consist of 2 different types of machines: scanners and targets. Scanner machines are used for launching scans. They contain all the required security testing tools. Target machines are the systems under test, i.e., they contain the latest product release candidate. Target machines should be kept clean to match production environment as much as possible. In other words nothing extra should be installed or configured. On the scanner machines though this is not as important, but limitations rather concern available memory and processing power needed by security testing tools. Figures 2.4 and 2.5 visualize both bare metal and cloud environments.

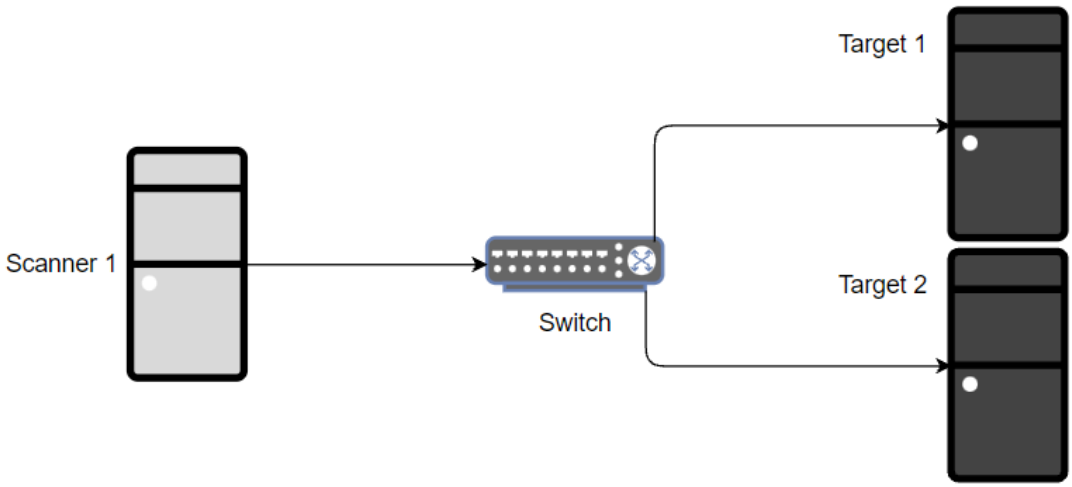

Figure 2.4. *Bare metal Topology*

Bare metal environment shown in figure 2.4 has one scanner machine and 2 target machines, while cloud environment shown in figure 2.5 has capability to have any amount of scanner and target machines. To not disrupt targets, only a single scanner should scan certain targets, hence multiple test setups in the figure. The only limitations come from cloud service's resource limits, namely RAM, CPU amount and storage space. Naturally all product types need to be included as targets and at least one scanner machine in order to test everything. However, the cloud environment also makes it possible to divide

test cases among multiple, identical targets.

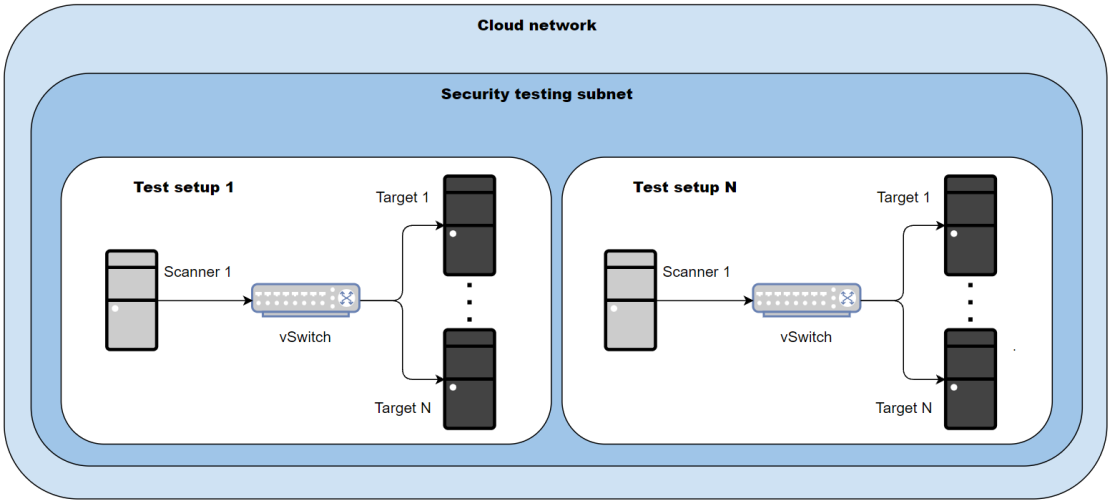

Figure 2.5. *Cloud Topology*

As can be seen, scanner and target machines are in direct contact (besides the switch). This way traffic is not interfered by other devices, such as routers. Certain security testing, mainly fuzz testing, could also harm these devices. Additionally, other devices could negatively effect execution times of test runs. Besides physical distance, another aspect to be considered in security testing is logical distance. This means the amount of security features between scanners and targets, e.g. firewalls. On the one hand target environment should be as similar as production environment, but on the other hand target should be tested in its most vulnerable state [36].

The product which is installed into the testing environments consists of 2 target servers with different roles. These product roles will be named type A and type B for this thesis. Product role A acts a master and product role B as a slave. There can exist multiple role B machines but only one role A in a single product deployment. Fuzz testing will be executed against both product roles but because role B machines are identical, there is only need for 2 target machines in the testing environments. However, it may be beneficial to have multiple identical targets in order to reduce load of a single target. Each product role is built out of several containerized components that communicate with each other over TCP/IP or UDP/IP. All web-based network traffic, namely GUI component traffic, uses HTTP and is secured with TLS.

2.6 Automated testing

Automated testing is a practice where test execution is left for program code. The objective is to minimize human interaction. The level of automatisation, i.e. how much responsibility can be left for program code, depends on the environment and nature of the tests. Simple tests that check a value of a variable probably do not require any hu-

man interaction. Security testing on the other hand might always require a human to filter out false positive results. Either way, test automation shortens tester's work effort [37] and is not dependent on working hours. Taking out the human factor also makes testing repeatable and uniform. In other words, tests can be executed in the same way each time.

Continuous integration (CI) is a software development practice, where code is frequently integrated into a central repository. This repository stores all the code and usually interacts with a runner application, that automatically verifies integrity of the software. Verification processes can vary a lot between projects, but generally software is automatically built and tested. Testing can also happen at several levels. For example, unit, system, performance and security are testing categories that could be part of CI. At the end there should exist some results to show whether quality gates are passed or failed. [38]

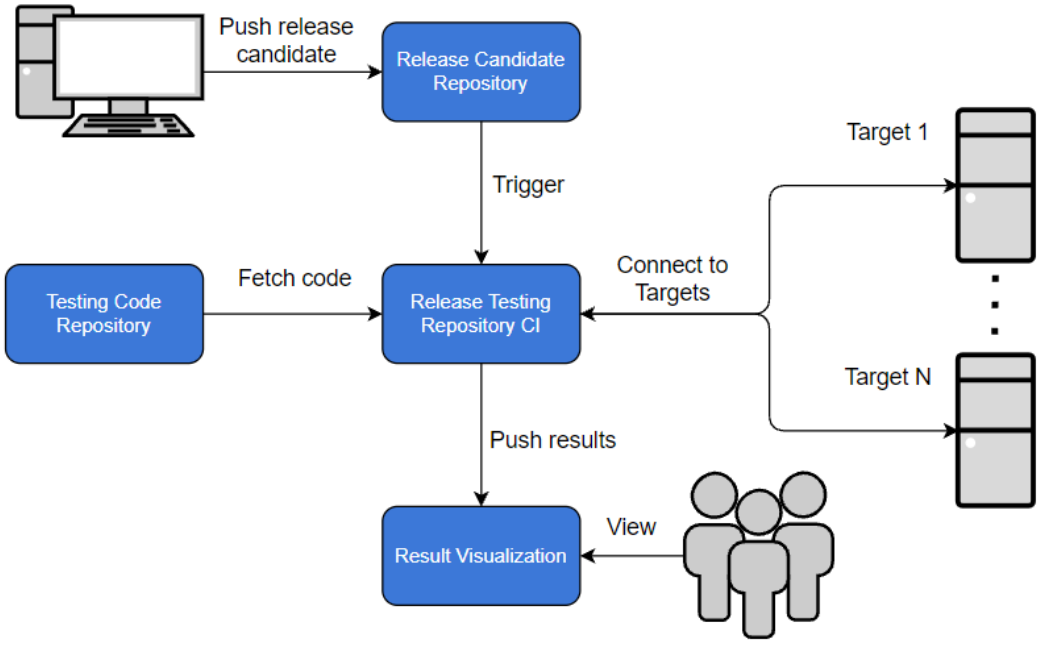

Figure 2.6. *General CI workflow in the given environment*

GitLab's built in tool GitLab CI/CD is used for continuous integration. It executes pipelines to verify software is ready for deployment. Pipelines comprises stages and jobs. A pipeline can have several stages, where each stage can have multiple jobs. Jobs tell the executing runner what to do, e.g. to run a certain script. In this scenario these scripts are either Robot framework commands which execute test runs or commands for a pipeline visualization tool. Jobs can also be dependent on previous jobs and they can contain conditional statements to decide if they are included in the pipeline, for example. Generally speaking, for a pipeline to successfully finish, all its stages and therefore jobs need to successfully finish. However, it is possible to make exceptions to this rule.

Product source code, testing code etc. are stored in multiple GitLab repositories. Se-

curity testing will be part of release testing repository. In other words, when ever a new product release candidate is published, all the security tests in the security pipeline will be run for that candidate. It would be possible to run security tests after each commit during development, but due to the time-consuming nature of security tests this would be impractical. The repository contains also other previously created pipelines, but they are not relevant regarding this thesis. Work is done only for integrating Defensics into security testing pipeline.

The CI workflow shown in figure 2.6 starts when a new product release candidate is pushed into release candidate repository. The release candidate repository then triggers CI in the release testing repository to execute its pipelines. Before executing the pipelines though, CI fetches testing code from testing code repository. Pipeline jobs execute Robot tests, which connect to one or more target servers via SSH. When testing is over, targets usually return some information back to CI server, which then assigns job verdicts. Finally, CI sends the results into a result visualization tool, where users are able to see all testing results.

Figure 2.6 is a high-level visualization and therefore hides details. One detail worth mentioning though is what happens when the security pipeline is triggered. First, the pipeline checks that the provided configuration files are valid. After that, installation job starts installing the product in the target machine. If installation succeeds, a sanity check is performed against the target. This essentially verifies that target is healthy and ready for testing. Finally, the actual testing can be started from the scanner machine. Once the tests have finished, it can be concluded that the product has either passed or failed the tests. To further provide visual feedback, job results after installation, sanity check and testing are sent into a tool that shows a run history of the pipeline for given release candidate build.

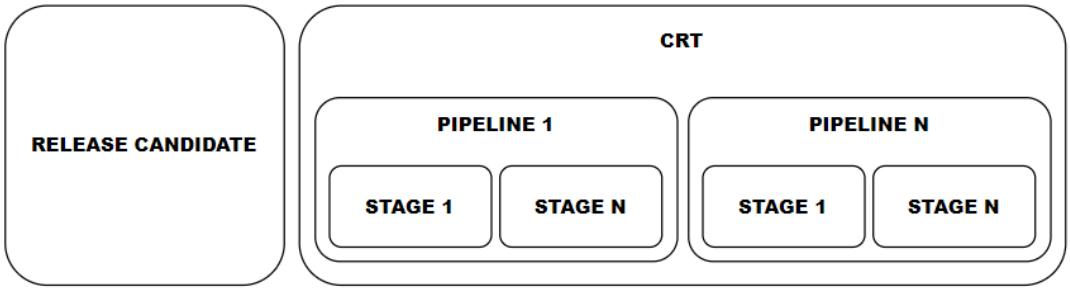

Figure 2.7. *CI Pipelines and CRT*

Figure 2.7 visualizes how different pipelines together form continuous regression testing (CRT), i.e., testing that new changes have not compromised software quality. The CRT is essentially what each new product release candidate is tested against. If all pipelines pass, the release candidate is ready for deployment. Likewise, if there are any fails, these

should be assessed. Security pipeline is one of the lines in the figure. The stages are previously mentioned steps, i.e. installation, sanity checking and actual testing. Stages can contain multiple jobs, but in the security pipeline each stage contains only a single job.

When considering the process that starts from actual programming and ends at CI results, there are multiple different technologies in play. The main technologies related to the CI environment of this work are Python, Robot framework and GitLab (CI/CD). GitLab is used as the repository manager for version control tool Git. Robot framework is a tool designed for test and process automation. In the given test environment it is used for calling Python methods, which execute the actual code for test cases. It should also be used for creating test cases for jobs in security pipeline(s). These tools and their purposes are listed in table 2.1 for clarity purposes.

Table 2.1. *Tools*

Tool	Purpose
Git	Version control system
GitLab	Git repository manager
GitLab CI/CD	GitLab's built-in tool for continuous software development
Robot framework	Generic open source automation framework for automated testing
Python	Most business logic for test execution

Robot framework has built-in libraries, but it also supports custom Python or Java libraries. Libraries located in remote servers and/or written in other languages can be used via remote libraries [39]. In the given environment custom libraries are written only with Python. Robot framework offers very human-friendly syntax, but as a consequence developing actual business logic is clumsier. Therefore, in the given environment Python is mainly used for creating business logic and Robot framework is used for wrapping the Python code into Robot keywords and test cases. This abstraction provides user friendly test cases even for inexperienced users.

Robot files, i.e. robot suites, are usually split into 4 sections: settings, variables, test cases and keywords. Settings is for importing libraries and other files and for defining metadata. Variables section is for defining variables, as the name suggests. Test cases include all runnable tests for the test suite and lastly, keywords can be used to call lower-level keywords or methods. Robot framework is not the main focus of this thesis, so external sources should be used for details. However, to give a basic idea of the syntax, program 2.1 is a simple robot test suite example. The program simply prints the string given in MESSAGE variable using a Python module. The Python module is not shown in the example, but it is only necessary to know that its print_message method has one

argument, which should be a printable string.

```
1  *** Settings ***
2  Library            UserMadePythonModule
3
4  *** Variables ***
5  ${MESSAGE}         Hello, world!
6
7  *** Test Cases ***
8  EXAMPLE_TEST_CASE_1
9      [Documentation]    Example test case
10     [Tags]  example
11     Print Message  ${MESSAGE}
12
13 *** Keywords ***
14 Print Message
15     [Arguments]  ${message}
16     UserMadePythonModule.print_message  ${message}
```

Program 2.1. *Robot framework example suite*

3. FUZZING

This chapter goes through theory behind robustness and fuzz testing. First definitions related to robustness are introduced, after which methods, classifications and basic concepts of robustness testing are discussed. Third subchapter is the core of this chapter: fuzz testing. It explains concepts, workflow, classifications and techniques related to fuzz testing. The subchapter covers many areas but leaves most technical details for other sources. Lastly, a separate part of fuzzing, network protocol fuzzing, is reviewed.

3.1 Robustness

ISO/IEC/IEEE 24765:2017 is a vocabulary designed to standardize software engineering terminology. It defines robustness as “degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [40]. Another definition provided by Avizienis et al. is “dependability with respect to external faults, which characterizes a system reaction to a specific class of faults” [41].

Avizienis et al. describe a system as an entity, that interacts with its environment. The environment consists of other systems. In this common environment, systems can act as providers, users or as both. Providers provide one or more services, that users receive [41]. Similar description can be found from ISO/IEC/IEEE 15288:2015, that defines system to be “combination of interacting elements organized to achieve one or more stated purposes” [42].

To define external fault, one needs to define a fault. Fault is the cause of error. An error exists when at least one of system’s external states differ from its correct state, which also implies a failure in the system. Failure means that system is unable to provide correct service. It is noteworthy that a fault does not always lead to an error, but error is always caused by a fault [22]. External state is a subset of system’s total state, which consists of 5 substates: computation, communication, stored information, interconnection and physical condition. The subset is determined by what can be perceived at the service delivery part of provider’s system boundary. [41]

External fault can harm a system only if there exists an internal fault in the system. If this kind of internal fault is related to any security properties, it is called a vulnerability

[22] [41]. ISO 27000 defines vulnerability as “weakness of an asset or control that can be exploited by one or more threats” [43], where asset means anything that can have value to a business. Same standard defines threat as “potential cause of an unwanted incident, which can result in harm to a system or organization”. Vulnerabilities and threats relate to assets: a threat can harm asset’s value by exploiting a vulnerability in its security properties [22].

By minimizing system’s internal faults, i.e. vulnerabilities, external faults, i.e. threats, have less surface to cause errors and therefore less failures in the system. Because system’s robustness was defined by how well it can function correctly in the presence of invalid inputs or stressful environmental conditions (i.e. threats), one can deduce that system’s robustness is also dependent on its vulnerabilities. The more vulnerabilities there are, the less robust system, and vice versa.

3.2 Robustness testing

Robustness testing is the act of finding vulnerabilities that negatively affect system’s robustness [44]. In a systematic literature review on software robustness conducted by Shahrokni et al. [45], it was found that fault injection is the primary method for robustness testing. Fault injection can be defined as the deliberate introduction of faults into a system [46]. Naturally the system behaviour must be observed afterwards to assess system robustness. In software fault injection faults are specifically inserted into a software system [47].

Based on study made by Benso et al. [48], fault injection techniques can be categorized into 4 classes:

- Hardware implemented
- Simulation-based
- Software implemented
- Hybrid

Hardware fault injection uses some physical device to change internal state of the SUT. For example, heavy ion radiation is a hardware-based fault injection method. In a simulation-based fault injection, an emulated model of the system is modified [45]. Hardware description language, such as VHDL, descriptions are examples of models that simulation-based fault injection can test. Software fault injection uses software to test the target system. Hybrid methods are combination of the above. [48]

In the context of software robustness testing, software implemented fault injection is the primary fault injection method. However, simulation-based techniques are also used in some environments [45]. In this thesis software robustness and therefore software fault

injection techniques are the only focus of interest. Hence, software robustness testing and software fault injection are referred mainly as robustness testing and fault injection from now on.

If a fault successfully manages to cause an error in a system, a failure may occur. These robustness failures can be classified using different benchmarks. The most common one is CRASH criteria [45][49], which categorizes failures using 5 stages:

- Catastrophic
- Restart
- Abort
- Silent
- Hindering

Catastrophic failure causes whole system to fail by crashing or rebooting. Restart failure leaves a single task in a state, where restarting is required. In abort failure a task terminates unexpectedly. Failure is silent when an error should be generated, but no such action is taken. Lastly, hindering failure generates an incorrect error. Generating a correct error is considered robust [44][49].

Fault injection and robustness testing in general can be done manually or by automation. However, in practice complete manual robustness testing is usually not feasible due to large amount of test cases. One of the most well-known [45] automated robustness testing tools is Ballista. Ballista uses software fault injection using both valid and invalid inputs. In the paper published in 1998 by Koopman et al. [50] Ballista was run against 233 function calls in 10 different POSIX operating systems. The results showed that approximately half of the functions were not robust. Failures were considered only on the first 3 stages of CRASH criteria.

Nowadays there exists multiple modern tools for automated robustness testing, but the tools are usually not referred as robustness testing tools. Instead, the automated tools are fuzz testers. Fuzz testing, or fuzzing, means sending malformed data to a system to measure its robustness [51]. It is therefore one fault injection and also robustness testing method. Although as pointed out by [51], the terms robustness testing and fuzz testing are sometimes mixed along with other terms. Fuzz testing is discussed more thoroughly in the next subchapter.

Software robustness testing methods can be categorized into several categories [44], but in practice many methods come down to invalid inputs. Invalid inputs can be intentionally chosen outside of their specific domains, or they can be randomly generated. Either way, conclusions can be made of SUT's robustness based on its reaction to invalid inputs. A fault injection method that is not solely based on invalid inputs is mutation testing. In

mutation testing code is changed by, for example, deleting function calls or by changing function call order [44]. Invalid inputs for functions are also mutation testing technique.

Besides invalid inputs, stressful conditions are another fault type. During stressful conditions, system experiences conditions that test its resource limits. If resources are exhausted, system may experience a failure. A robust system should be able to not exhaust all of its resources. Hence, it is important to include stressful conditions as a part of robustness testing. An attack type that is based on resource exhaustion is called denial of service (DoS). Testing for DoS scenarios can happen via fuzz testing [45].

3.3 Fuzz testing

Fuzz testing – a.k.a. fuzzing – is a robustness testing method, which can be used as a part of security testing. In particular, the simplicity and efficiency of fuzzing have made it increasingly popular option for security testing [52]. This subchapter first defines basic concepts related to fuzzing and proceeds to describe the general workflow and algorithm. Next, different types of fuzzers and their differences are discussed. Lastly, various techniques used during fuzzing are listed and explained.

3.3.1 Basic concepts

In 1990 Miller et al. [53] published a paper called "An empirical study of the reliability of UNIX utilities". In the paper a program called fuzz was used to generate random characters to test reliability of several Unix utilities. As a result, 24%-33% of utility programs in different systems experienced a crash or were left in hanging state. From this simple yet seemingly effective program the currently used terms such as fuzzing, fuzzer and fuzz testing were derived [12].

Fuzzing is “a highly automated testing technique that covers numerous boundary cases using invalid data ... as application input to better ensure the absence of exploitable vulnerabilities” [54]. We can rephrase this definition using definitions from earlier: fuzzing is an automated robustness testing technique that uses invalid input injection. Fuzzer, or fuzz tester is a tool that generates the invalid inputs. Or by definiton given by [55]: “a tool that tests a target program by iteration and random input generation”.

Some papers (e.g. [54]) do not define fuzzing as generating invalid data, but rather generating semivalid data, i.e. almost valid data. This is an important distinction. If SUT receives data that is completely invalid, it might be dismissed immediately. However, if the data is semivalid, the SUT is able to accept it, which may eventually lead to a failure [54] [56]. Hence, semivalid data is usually the preferred option for a fuzzer.

The following chapters and subchapters contain various terms related to fuzzing. For

Table 3.1. *Fuzzing related terminology*

Term	Explanation
Fuzzing	Automated robustness testing method for finding vulnerabilities by means of injecting invalid (or semivalid) inputs
Fuzzer	Tool that performs the fuzzing
Fuzz run	A single injection including the result evaluation
Fuzz configuration	Parameters that control a fuzz run, e.g. PUT and current seed
Fuzz campaign	Complete fuzzing execution, i.e. a set of fuzz runs
Seed	Valid data that is modified during fuzzing
Test case	Invalid (or semivalid) input, e.g. a modified seed
PUT/SUT	The program or system that fuzzer is run against
Mutation	Seed modification

clarity, table 3.1 has a collection of terms that will be used throughout this thesis. The table is also useful because some papers have slightly different meanings for the terms. Regardless, the attempt was to use the most popular explanations.

Some papers (e.g. [1]) differentiate fuzzing from fuzz testing: fuzzing is considered the actual execution while fuzz testing utilizes fuzzing to test security requirements of the PUT/SUT in question. However, generally – including this thesis – fuzzing and fuzz testing are used interchangeably.

3.3.2 Workflow

Fuzzing workflow describes what sort of fuzzing strategy a fuzzer uses. For example, how inputs are generated and how are SUT reactions evaluated. Because there are many different types of fuzzers, fuzzing strategies can differ vastly. Hence, it is impossible to give detailed description of a workflow that applies for every fuzzer. However, a higher level structure can be given. Even for this though there does not seem to exist a standardized model. Instead, papers give their own interpretations of a general fuzzing workflow. Figure 3.1 visualizes one type of workflow. It is a simplified adaptation of the figure in [57]. There are 4 main components in play: test case generator, SUT, static/dynamic analysis and vulnerability detector. Test case generator generates test cases and sends them to the target system. Static and dynamic analysis techniques can guide the fuzzing by extracting additional information from SUT. The SUT reactions are then evaluated using vulnerability detector, which reports all findings.

Vulnerability detector can also be called a bug oracle, or oracle for short. It is the part of fuzzing that decides whether PUT/SUT has reached an erroneous state [1]. At the sim-

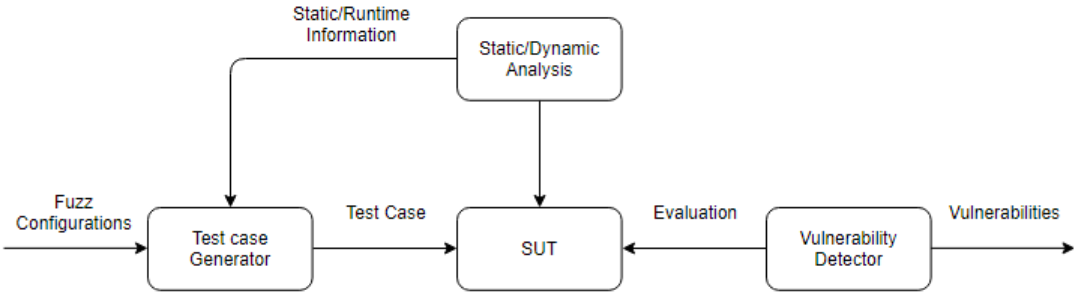

Figure 3.1. *Fuzzing Workflow*

plest case, an oracle only observes crashes. However, more advanced fuzzers, namely grey box and white box fuzzers, are also able to detect less serious faults. The challenge of observing PUT/SUT reaction to generated input and deciding if a fault exists, is generally known as *test oracle problem* [58]. Valid case evaluation is a method where a valid case is sent to SUT to see if it still responds as expected. This allows detecting catastrophic failures, but not less serious failures. If SUT can be accessed, its resource usage can be monitored and protocols such SNMP can be utilized [59] to expose even minor failures. Advanced evaluation includes methods such as dynamic binary instrumentation, running source code in a debugger or analyzing memory allocations [59]. In some cases it is possible to check correct functionality of target using case specific evaluation methods. For example, complex network protocol implementations such as TLS can be tested by providing invalid credentials [59].

The original figure mentions only a program, but same workflow can be generalized to systems. Initialization phase of the algorithm is not included in the figure because SSP (see below) is included in test case generator and other methods vary a lot between fuzzers. Scheduling and updating phases are also included in test case generator. Static and dynamic analysis might not be used at all, especially for black box fuzzers. For grey and white box fuzzers though there should be some level of analysis. Another note about the figure concerns vulnerability detection: all detected errors might not actually be vulnerabilities. They could be security-wise harmless bugs or even false positives.

Fuzzing workflow can also be described as an algorithm. Manès et al. [1] introduced an algorithm consisting of 6 procedures, which have been simplified to 5 phases described below:

- Initialization
- Scheduling
- Generation
- Evaluation

- Updating

First, some fuzzers need to be initialized. There are multiple options that can be done here. One possibility is to insert instrumentation code into SUT, that will be used during fuzzing as a health check. It is also possible for a fuzzer to capture the program/system state, before starting the fuzzing. This can be useful in situations where the PUT/SUT takes time to initialize itself before accepting input. This technique is called in-memory fuzzing. [1]

A problem that might need to be solved during initialization phase is seed selection problem (SSP). The problem is that there can be large or infinite amount of valid seeds to be used for fuzzing, but only finite amount of time. Furthermore, seeds often produce duplicate bugs, but for optimal efficiency no duplicate results should occur. SSP means selecting minimal amount of seed(s) that maximize amount of bugs found, i.e., discarding redundant seeds. In addition, seed size might be optimized during initialization phase for minimized memory usage. Creating a driver program for indirect fuzzing can also be part of initialization phase. [1] [60]

Scheduling happens between initialization and test case generation. In fuzzing this phase means selecting a fuzz configuration for the next iteration [1]. The goal is to select a configuration that results in the most valuable outcome. In practice this means maximizing amount of vulnerabilities found. Because initially it is impossible to say which configuration results in most vulnerabilities found, each configuration should be fuzzed enough to find out whether the configuration is valuable. However, the caveat is wasting time on bad configurations. This inherent problem in scheduling is called fuzz configuration scheduling (FCS) problem [10].

Simple fuzzers might not need scheduling phase at all, but usually it makes sense to have a scheduling algorithm to guide the fuzzing process. The information used for decision making varies between fuzzers. Most notably, fuzzer type affects the amount of available information. For example, black box fuzzers cannot use code coverage or constraints collected during symbolic execution as their information, unlike grey box (e.g. [61]) and white box fuzzers (e.g. [62]). Rather, black box fuzzers need to rely on simpler information, that is observed crashes and execution time, for instance [10].

After fuzzer has been initialized and a fuzz configuration has been selected, the actual test cases, or inputs, can be generated. There exists two primary approaches for this phase: mutation- and generation-based fuzzing [1]. Mutation-based fuzzers generate inputs by modifying the system's valid data, i.e., seed data, and then feed the modified data into SUT. Generation-based fuzzers on the other hand use a complete specification of the system's inputs and generate the test cases based on that. The fuzzer type affects how much information can be used for generating the test cases. For example, white box fuzzers have more techniques available than black box fuzzers.

To determine if a test case has been successful, there needs to be a way to observe SUT reaction. This is the evaluation, a.k.a instrumentation, phase. How the evaluation is done heavily depends on the fuzzer and PUT/SUT. At the simplest cases evaluation can mean making a connection attempt to SUT or observing any segmentation faults generated by PUT. If connection cannot be established, or a segmentation fault is produced, it can be said with high certainty that PUT/SUT has crashed. However, if there is a need to observe less significant errors, evaluation becomes more case-by-case matter and likely more challenging.

Knowing the cause of a fault is a crucial part of fuzz testing. After all, the reason why fuzz testing is performed is to expose vulnerabilities that should be fixed. Without understanding the cause, fixing the vulnerability becomes unreasonably hard task. This is why instrumentation is such an important stage. As shown later on, instrumentation might not happen after every test case. Instead, instrumentation could be after certain period of time. This can save time when instrumentation takes a lot of time, e.g. with network protocol fuzzing. The drawback is that it becomes harder to identify which case caused a failure, if a failure occurs that is.

When a test case has been executed and its result evaluated, the fuzzer decides if it discards this configuration or keeps it for later use. In practice this means adding the configuration into a pool of configurations, that can be used for fuzzing. During updating phase it is also possible to remove configurations that are not expected to provide value anymore. Selecting which configurations to keep in the pool introduces a similar problem as SSP in initialization phase: only minimal amount of configurations that result in maximal vulnerabilities should be kept in the pool. Instead of adding and removing configurations it is also possible to prioritize certain configurations. [1]

Once again, fuzzer type affects the execution of updating phase. The amount of information available on PUT/SUT determines how sophisticated decisions can be made on value of the configuration. Usually black box fuzzers do not update their configurations since they have minimal amount of information to work with. Grey and white box fuzzers on the other hand have information available, which they can use to update their configurations. For example, if a configuration has discovered a new path, it could be a reason to keep it. [1]

3.3.3 White, grey and black box fuzzing

In software testing generally, 3 testing methods can be differentiated by their knowledge of the SUT. These methods are called black box, white box and grey box testing. According to NIST, the 3 methods can be defined as follows: black box testing has no information about internal structure or implementation details of the assessed object. White box testing on the other hand has all this information. Grey box testing is somewhere in

between the former two [63]. Assessed object is the PUT/SUT in the context of fuzz testing.

When talking specifically about fuzzing, the 3 methods can also be differentiated by their observation of the PUT/SUT, source code dependency and degree of program analysis [13]. Black box fuzzers only observe if program has crashed and they do not have access to source code or structure of the PUT. White box fuzzers can observe the source code itself, along with PUT reaction to test cases. They are able to systematically analyze the state space of the PUT by analyzing its structure and runtime information [1]. Grey box fuzzers are again in between the former: they can gather some intermediate information about the SUT [55]. The information is anything that helps generating test cases, such as code coverage or memory usage [12]. Code coverage measures the amount of execution paths covered, i.e. executable control flows of a program. For example, a simple if/else statement offers two possible execution paths. The distinctions between the 3 types of fuzzing methods are not always clear, as pointed out by [1]: some black box fuzzers collect some run information and white box fuzzers may have to rely on some approximations.

The 3 methods each have different advantages and drawbacks, so appropriate method is very context dependent. White box fuzzing can offer high code coverage by systematically traversing code and detecting boundary conditions and possible vulnerabilities [64]. However, the drawback is that it consumes more time and space than black and grey box fuzzing [55]. Slow execution comes often from the overhead of analyzing every command of PUT [1], but also building a data structure of explored paths takes time and memory [61]. There have been attempts to minimize the drawbacks by manual guidance or by including grey box fuzzing features [1]. Despite these attempts white box fuzzing is not regarded as practical currently [12].

Black box fuzzing can be very effective due to its simplicity. It fits well in a situation where efficiency matters over result quality [12]. This is because black box fuzzers are able to produce test cases faster than white and grey box fuzzers [65]. Because black box fuzzers do not have access to observe SUT other than input and output, they cannot observe non-critical failures. Hence, black box fuzzer results almost certainly do not contain false positives. Sometimes the advantage of black box fuzzing is that other techniques are not viable. After all, white and grey box fuzzers require at least some information about the system. In some cases no such information can be gathered and black box fuzzing is left as the only option.

The biggest drawback of black box fuzzing is low code coverage [12]. Low code coverage leads to undetected vulnerabilities. Especially random mutation-based black box fuzzers are incapable to reach high code coverage due to their simplicity [12]. A concrete example is the area of web applications. Several papers have been published (e.g. [66], [67], [11]), that show inefficiency of black box web scanners. These scanners operate like fuzzers,

as they try to detect web-based vulnerabilities and inject faults to cause a failure. In [66] 11 web scanners were tested against an artificial web application. The results show that some vulnerabilities were not found by any scanner and none of the scanners managed to find over 50% of the vulnerabilities. These results cannot be generalized to all black box fuzzers, but they do show that black box fuzzing can be inefficient in some cases.

Peach fuzzer is a popular black box fuzzer [68] [69]. It is a model-based fuzzer as its seed generation is based on models, called Peach Pits [57]. Peach Pit files are XML files that contain all needed information for running fuzz tests. This information is mainly seed data structure and instructions for sending and receiving data. Creating the Peach Pit files, or modeling, is the most time-consuming part because more detailed models create higher quality test cases. According to developers of Peach, quality of the files is what differentiates a "dumb" Peach fuzzer from a "smart" Peach fuzzer [68]. As we will see later on, this is typical for model-based, a.k.a generation-based fuzzers.

To address the challenges of black and white box fuzzers, grey box fuzzers try to have the best of both worlds. Mainly, grey box fuzzing tries to improve low code coverage of black box fuzzers and reduce time spent on program analysis and constraint solving that white box fuzzers suffer from [61].

American Fuzzy Lop (AFL) is an example of a popular and efficient grey box fuzzer [13]. To be specific, AFL is a coverage-based grey box fuzzer. It has found hundreds of vulnerabilities, including popular tools such as bash, curl and openssl [70]. AFL is probably one of the most common fuzzers mentioned in literature. In a literature review in 2018 Klees et al. found that almost half of the papers they collected to evaluate fuzz testing compared their own fuzzers against AFL [55]. Several fuzzers have been derived from AFL, such as

- AFLFast: AFLFast explores new paths faster than AFL by using a Markov chain model and focusing on the low-density areas of the model. Indeed, on average AFLFast exposed an error 19 times faster than AFL in results provided by AFLFast developers. It also found a few erros not found by AFL [61]. However, AFLFast does not provide significant advantages in some target programs, such as image processing programs tested in [55].
- AFLSmart: Generation-based fuzzer that combines input structure component of Peach and coverage feedback of AFL. Proved to provide better code coverage and amount of zero day bugs found compared to AFL and AFLFast in some open source libraries. It also outperfomed VUzzer, although VUzzer did find a bug that was not discovered by AFLSmart. [69]
- VUzzer: Uses static control and data flow analysis in combination with dynamic taint analysis to produce fewer higher quality inputs. Proved to find significantly more unique vulnerabilities in real-world programs than AFL while using less time

to do so. [71]

In a study made by Pham et. al. in 2019 [69], grey box fuzzing was compared against black box fuzzing. In particular, code coverage and number of identified bugs were measured. AFL was one of the used grey box fuzzers and Peach was the only black box fuzzer. In the study Peach was used as a generation-based fuzzer rather than mutation-based fuzzer. The results show that AFL outperformed Peach in almost all of the tests. For instance, AFL found 16 zero day bugs, while Peach only found one. The researches explain the results by the lack of feedback coverage and possibly insufficient seed specification in Peach.

Although some results in fuzzing studies, even mentioned in this thesis, might indicate one fuzzer being better than the other, readers should not make generalized conclusions. As Klees et al. [55] have shown: “In general, few papers use a common, diverse benchmark suite”. Hence, comparisons between fuzzers are hard to make accurately. In order to make fair comparisons, studies should use same datasets, similar environment and same fuzz configurations.

3.3.4 Mutation- and generation-based fuzzing

Another way to classify fuzzers is by their input generation method. There are 2 primary methods for this, already mentioned in this paper: mutation-based and generation-based fuzzing [57].

Mutation-based fuzzers often need seed data, which they modify, i.e. mutate, and then send the modified data to SUT [56]. The simplest mutation-based fuzzers do not need any seed data because they generate inputs purely randomly. However, this kind of random fuzzing is ineffective. A classic example is to consider an if statement that compares input to some integer n. If integers are 32 bit, a fuzzer based on purely random inputs only has probability of $\frac{1}{2^{32}}$ to execute that path. More than likely the random input will get discarded by PUT.

Purely random fuzzing is clearly ineffective, which is why mutation-based fuzzers need some intelligence to guide the fuzzing. The intelligence comes from seed data. The seed data is valid data for PUT, which is then slightly modified in order to detect vulnerabilities. Modifications can happen at bit level by, for example, flipping, deleting and adding bits. Some fuzzers may also use common problematic values in their operations, such as 0 and -1 for integers. The main goal is always to modify seed data enough to cause problems in PUT/SUT, but not too much so input is not instantly rejected. [1]

Generation-based fuzzing does not require seed data, but a specification, that is either manually (e.g. Peach [68]) or automatically analyzed (e.g. PULSAR [72]). Based on the analysis, semi-valid data is produced to test PUT/SUT [56]. Specification describes

complete input format for PUT/SUT, which makes it possible for a fuzzer to generate high quality test inputs. This, however, makes the fuzzer dependent on the quality of the provided specification. Poor specification may produce poor results. Specification format depends on the fuzzer. Network protocol fuzzer SNOOZE [73] and all-around fuzzing framework Peach [68] both use their own XML-based models. Models can be written based on completely new protocols, or from RFC specifications, for instance. As another example, kernel API fuzzer Trinity [74] uses C structs to describe arguments of system calls.

Mutation-based fuzzing is simpler than generation-based fuzzing. It only requires seed data, which can be a file or any data accepted by the fuzzer. Seed data could be given manually (e.g. AFL [13]) or fuzzer may retrieve it automatically by capturing network traffic (e.g. LZFuzz [9]), for example. After that, fuzzing works similarly despite the input format. Seed data can be modified randomly or by applying certain techniques, such as (dynamic) symbolic execution or (dynamic) taint analysis [75]. More about these techniques will be covered later on. If PUT performs complex input checking, mutation-based fuzzing most likely fails to pass the syntax checks [75]. This is due to limited understanding of the correct input format.

Generation-based fuzzing requires studying the specification to generate test cases. This is often time-consuming, especially when done manually [57]. The advantage of generation-based fuzzing is higher code coverage. This is due to deep understanding of a specification, allowing detection of hard-to-reach execution paths. This indeed makes it easier to pass syntax checks, but even generation-based fuzzers may struggle with semantic checks [75]. In a 2007 study that compared code coverage of mutation and generation-based fuzzers against PNG file [56], generation-based fuzzer covered 76% more code than the mutation-based fuzzer.

A survey from 2019 compared a large set of fuzzers [1]. From the results it can be seen that mutation-based fuzzers are more popular than generation-based, also known as model-based, fuzzers. Another survey from 2018 backs up this claim [13]: “mutation based fuzzers are easier to start and more applicable, and widely used by state-of-the-art fuzzers”. However, most generation-based fuzzers listed are developed in recent years, which indicates that generation-based fuzzing has not received as much research as mutation-based fuzzing. Choosing between mutation- and generation-based fuzzing is context dependent, but generally if PUT/SUT has specific or strict input format (e.g. compiler or network protocol), generation-based fuzzing is more effective [12].

3.3.5 Techniques

Even if two fuzzers are classified as white box or generation-based fuzzer, their underlying algorithms can differ. More precisely, the algorithms that are used as part of test case

generation. In a systematic review of fuzzing techniques in 2018 [57], Chen et al. present various techniques they have discovered to be used during fuzzing to improve efficiency. Furthermore, they have categorized these techniques into 3 categories by their role in fuzzing workflow: sample generation techniques, dynamic analysis techniques and static analysis techniques. Sample generation techniques determine the core strategy of test case generation. Chen et al. categorize these into random mutating, grammar representation and scheduling algorithms. In practice these correspond to mutation-based fuzzing, generation-based fuzzing and scheduling algorithms discussed earlier.

Dynamic and static analysis techniques are used to guide the test case generation based on runtime and static information, respectively. Dynamic analysis techniques observe a running program either in a real environment or in an emulator [14]. By gathering runtime information they can gather accurate and context specific data, which can be used when generating inputs. Static analysis techniques on the other hand observe a static program, e.g. source code. Because static analysis does not have knowledge of runtime information, it has tendency to be less accurate and produce more false positives than dynamic analysis [14]. However, static analysis is more effortless in a sense that the PUT does not need to be run.

Both dynamic and static analysis techniques are only used in white or grey box fuzzers because they rely on information gathered from PUT/SUT. Especially dynamic symbolic execution and dynamic taint analysis are techniques used in white box fuzzers [1]. Both techniques are briefly introduced below and listed under corresponding categories. In addition, fuzzing based on machine learning is introduced in the last sections. All techniques are only briefly described. Detailed explanation of the techniques is beyond the scope of this thesis.

Dynamic analysis techniques

- **Dynamic symbolic execution**: To discover most paths possible, dynamic symbolic execution (DSE) traverses the program using both concrete values and symbolic expressions [76]. Ideally, all execution paths and inputs that satisfy those paths are found, which then makes vulnerability testing possible. DSE has a few problems [14], but the most prevalent problem is path explosion. This means the number of possible execution paths, or branches, grows exponentially, making the testing infeasible [77].
- **Coverage feedback**: As program is traversed, input effect on the program under test is observed. If the fuzzer decides the input caused an interesting reaction or a failure, the input is stored and mutated for further testing. Otherwise it is discarded. What is considered as interesting is dependent on the fuzzer. [61]
- **Dynamic taint analysis**: Input data is marked with metadata, or *tainted*, and the execution path is tracked during runtime. When the tainted data changes another

data, it also becomes tainted. As a result, all affected data can be tracked, which helps in detecting vulnerabilities. [78]

Static analysis techniques

- **Control flow analysis**: Control flow analysis (CFA) is a static way of trying to discover execution paths of a program. The instructions of a program are used to produce a tree structure, called control flow graph (CFG). Based on the CFG it is then possible to send semivalid data to execute certain CFG paths with the ultimate goal of discovering vulnerabilities. [79]
- **Data flow slices**: The program under test is stripped, or *sliced*, so that only certain data and all the data affecting it are left. The purpose is the same as with control flow analysis, and CFG can also be used for slicing. [57] [80]

In addition to dynamic and static analysis techniques, machine learning-based fuzzing is a newer approach. Machine learning means program improving its performance at some specific tasks after being exposed to an experience [81]. In practice, learning happens by iteration: program gathers experiences by performing similar tasks and hence improves its performance in those tasks. The experience is data that the program has received from some source [82]. For fuzz testing, those experiences can be things such as choosing semivalid data to bypass format check of a program [14].

In a systematic review of machine learning-based fuzzers done by Wang et al. in 2020 [14], it was found that machine learning-based fuzzers can be more efficient than traditional fuzzers. For instance, code coverage was improved by 17.3% on average compared to traditional fuzzers. However, machine learning-based fuzzers are not (yet) better at everything, as the study shows: no significant differences can be found in number of found vulnerabilities in a popular LAVA-M dataset [83]. The reasoning behind this result is that the vulnerabilities are deep in the dataset's program code and traditional fuzzers still have better program analysis capabilites. In general, machine learning-based fuzz testing is still in research stage, but the growth in the amount of research papers and proved advantages over traditional fuzzing methods show that it is a viable and effective method [57] [14].

It is noteworthy that fuzzers do not have to be purely one type. Instead, they can execute steps in the workflow using methods from different fuzzer types. For example, SymFuzz runs fuzz campaigns as a black box fuzzer, but before that it gather information from SUT much like a white box fuzzer [84]. To be more precise, it first tries to find the optimal mutation ratio, i.e. ratio of modified bits to total size of seed, for a program-seed pair. SymFuzz then uses the mutation ratio to generate test cases, without probing the SUT anymore. Driller is another example: it starts executing as a grey box fuzzer, but proceeds to use dynamic symbolic execution — a distinctive white-box technique — when hard to

reach paths are found [85].

3.4 Protocol fuzzing

RFC 1208 [86], or A Glossary of Networking Terms, defines protocol as “A formal description of messages to be exchanged and rules to be followed for two or more systems to exchange information”. A computer network on the other hand is a collection of computers capable of changing information [87]. Network protocol is therefore a formal description of messages and corresponding rules for computers to exchange information.

The actual descriptions of a network protocol are described in a protocol specification [88]. Protocol specifications include information about information change methods, behaviour and network packet format. In practice, these specifications are usually Request for Comments (RFC) documents, which also cover other aspects of computer networking [89]. An example is Transmission Control Protocol (TCP), which has its protocol specification described in RFC 793 [90].

To actually use a network protocol, it needs to be implemented. Implementations are built from the protocol specifications by manual interpretation. As the specifications may be too vague, or they are misinterpreted, implementation errors can happen [88]. Furthermore, implementation errors can be hard to detect and in the worst case scenario, these errors are security vulnerabilities that can be exploited later on [22].

Undetected vulnerabilities in protocol implementations can cause serious problems. Heartbleed is an infamous example of a serious protocol implementation vulnerability [91]. It affected TLS/DTLS implementations of early OpenSSL versions, and made it possible to read private memory locations, possibly exposing private keys and passwords. As OpenSSL is a popular library, the effect was widespread. Durumeric et al. [92] found that 24-55% of the one million most popular HTTPS-enabled websites were affected by Heartbleed. Similarly, other network protocols impose a risk of containing widespread consequences due to their heavy usage [52].

It is evident that there is a need to detect these vulnerabilities before they are implemented. Therefore, it is important to conduct proper testing for the implementations. This is where automated tools can provide help. As discussed earlier, fuzzing can be used for verifying software security and this applies for network protocols too. Same classifications, methods and techniques apply. However, network protocol fuzzing has some unique aspects to consider.

Fuzzing network protocols introduces a couple new challenges to fuzzing. As protocols are mostly implemented through state machines, stateful fuzzing is required. This, however, is often not the case with fuzzers. Instead, fuzzers are usually stateless. Second challenge arises from the fact that vulnerability in a protocol may require a specific com-

bination of substates. The fuzzer needs to "lock in" a certain substate thats lead to the vulnerability before moving on to the next substate. In practice, locking in the states means first generating valid inputs to reach deeper states [93]. After that invalid inputs can be generated to exploit vulnerabilities. Stateless fuzzers that are able to fuzz one packet at a time are inefficient in finding these deep paths. [52]

Also, some protocols, such as security protocols, may use encryption for the messages. In these cases fuzzers should be able to decrypt the messages. Otherwise fuzzed messages could be completely invalid and hence instantly rejected by the SUT. Security protocols also impose other challenges. For instance, keys that ensure message freshness mean that fuzzers cannot use previous messages as inputs. Instead, fuzzers must use fresh messages by acting as a man-in-the-middle, for example. [93]

Many protocols are based on client-server model [59]. Therefore, an aspect to be considered in protocol fuzzing is whether SUT acts a server and fuzzer as a client or vice versa. The former is the simpler case as fuzzer only needs to initiate connections and observe SUT reactions. When fuzzer acts as a server, it needs to listen for connection attempts and respond with malformed messages. Furthermore, the SUT acting as the client needs to repeatedly initiate connections for fuzzing to continue. This may require modifications to the SUT.

In the realm of protocol fuzzers, generation-based fuzzers need a protocol specification to generate semi-valid data. This implies that certain semi-valid inputs can only be applied to certain protocols. New data must be generated for each protocol. By contrast, mutation-based protocol fuzzers examine a current session and always apply same methods for generating semi-valid data. Hence, mutation-based fuzzers are generally simpler. [15]

Mutation-based protocol fuzzers are more common than generation-based protocol fuzzers [1]. Examples of mutation-based protocol fuzzers are AutoFuzz and SecFuzz [94] [93]. Both of these act as a man-in-the-middle between a client and a server to examine messages and produce semi-valid data. SNOOZE and PULSAR are examples of generation-based protocol fuzzers [73] [72]. SNOOZE's fuzzing engine takes a manually made specification as an input parameter. PULSAR on the other hand generates the specification model automatically, by inferring network traffic. From the model it can deduce the protocol state machine and message formats. This information is used to guide the fuzzing later on.

Generation-based network protocol fuzzers, like all fuzzers, can get specification, also called as model, manually or automatically [1]. Manual acquiring happens by predefined models or by user in a specified format. SNOOZE is an example of this kind of fuzzer. User can use either predefined models or use their own model in XML format. Conversely, inferring the model automatically requires fuzzer to have access to previous messages where the protocol was used, or to live traffic. PULSAR is an example of this kind of

generation-based fuzzer, as it infers a protocol model from live traffic.

SUT is often treated as a black box in network protocol fuzzing, meaning that many fuzzers only analyze the output of SUT by comparing response to valid and invalid messages [15]. However, it is possible to conduct deeper analysis to identify the exact vulnerabilities. A fuzzer called SecFuzz uses dynamic analysis tools to detect memory errors [93]. In the SecFuzz study, a use-after-free memory access vulnerability was found. The used dynamic analysis tool showed a stack trace which made it easy to detect cause of the vulnerability. However, as it usually is with white box techniques, dynamic analysis adds overhead, which in some cases can be significant.

Protocol fuzzers can work in either message-level or state-level [15]. Message-level fuzzers alter individual messages, but the order of protocol messages is kept intact. State-level fuzzers on the other hand can also alter the message order. Message-level fuzzing is the more common option [95]. Message-level and state-level fuzzing are also known as bit-level and order-level fuzzing, respectively [95]. AutoFuzz is an example of a message-level fuzzer [94]. It captures live messages, constructs a model of them and performs fuzzing operators to the models. AutoFuzz is not able to modify message order though like a state-level fuzzer. An example of a state-level fuzzer is SecFuzz [93]. It is able to insert a message from a previous session to random position of current session's message sequence. SecFuzz is also able to alter structure of single messages.

A recent network protocol fuzzer is AFLNet [96], which extends the previously discussed grey box fuzzer AFL. AFLNet combines coverage-based grey box fuzzing (CGF) and stateful black box fuzzing (SBF) into stateful coverage-based grey box fuzzer (SCGF), where fuzzing is guided by server response codes. It addresses the problems in CGF and SBF. The problems mainly being limitations in stateless nature of CGF and SBF's dependency on protocol specification quality. Unlike most network protocol fuzzers, AFLNet is a mutation-based fuzzer. Also, unlike regular mutation-based fuzzers, seeds are message sequences instead of individual messages. These sequences are mutated on both state- and message-level. The results of the study suggest that AFLNet can outperform both a CGF and a SBF fuzzer in coverage and vulnerability finding.

3.5 Fuzz testing maturity model

Fuzz testing varies a lot between different entities because the used tools, environments, targets etc. differ. It is therefore hard to compare how well fuzzing has been done between all these entities. Fuzz testing maturity model (FTMM) [97] attempts to provide a solution for this by providing an unified framework to measure maturity of fuzz testing, independent of all factors affecting fuzzing. The framework uses numbers from 0 to 5 for describing the maturity level. Each level contains progressively more requirements that must be met. The 6 different levels and their brief descriptions are listed in table 3.2. FTMM states that

Table 3.2. *Fuzz Testing Maturity Model Levels*

Level	Mut/Gen	Mut. min. effort	Gen. min. effort
0 - Immature	One	<2 h / <100k cases	<2 h / <100k cases
1 - Initial	One	2 h / 100k cases	2 h / 100k cases
2 - Defined	One	8 h / 5M cases	8 h / 1M cases
3 - Managed	Both	16 h / 5M cases	16 h / 2M cases
4 - Integrated	Both	1 week / -	1 week / -
5 - Optimized	Both	2 * 30 days / -	2 * 30 days / -

“The test case counts and minimum testing times in this document are based on years of industry averages”.

Level 0 means no fuzzing has been performed or the fuzzing is less than required by level 1, which requires 2 hours or 100k test cases using either mutation- or generation-based fuzzer. Level 2 ramps these numbers up to 8 hours or 1 million test cases for a generation-based fuzzer and to 8 hours or 5 million test cases for a mutation-based fuzzer. Level 2 also requires a full attack surface analysis, while level 1 defines the requirements only for known attack vectors. Level 1 also allows assertion failures. Levels 2 and 3 only allow transient failures.

From level 3 onward, fuzzing must be performed using both mutation- and generation-based fuzzer against full attack surface. Level 3 requires 16 hours of fuzzing for both types, or 2 million and 5 million test cases. It is also the first level where automated instrumentation must be used and test configuration must be documented. Level 4 increases fuzzing time to 1 week for each fuzzer and removes the option for running certain amount of test cases. In addition, fuzzing must be part of automated testing and component analysis must be performed, which means analyzing SUT for components, e.g. third-party libraries. Level 4 does not allow even transient failures. Level 5 is the highest level of FTMM and the minimum fuzzing time is 30 days using 2 mutation-based and 2 generation-based fuzzers. In addition to level 4, SUT must be analyzed to detect subtle failures and code coverage must be measured, i.e. white-box techniques should be used.

Much like CRASH criteria, FTMM defines different types of failures. What is special to the model though, is that it defines two types of failures that are allowed at some maturity levels. These are assertion failures and transient failures. Assertion failures cause some internal check to fail in the target software, but do not produce an error which would stop execution. Transient failures on the other hand might be legitimate failures during initial run, but they are considered transient if they cannot be easily reproduced.

When measuring maturity level of a target, the whole attack surface must be taken into account, which means that in practice attack surface analysis is required even for level 1.

As with security in general, target is only as strong as its weakest link, i.e. attack vector. The maturity model of a target is determined by the attack vector with the worst maturity level. Thus, if at least one vector is not fuzzed, the maturity level of the whole target is 0.

4. DEFENSICS

This chapter introduces and analyzes Defensics, a fuzz testing tool focused on network protocols. First, Defensics is introduced by discussing its basic concepts and functionality. Then, fundamental requirements, namely installation and licensing, are gone through. Third, different settings for configuring Defensics are listed. Lastly, current challenges faced by the team using it are presented. As the main objective of this thesis is automation, this chapter primarily discusses things related to automation process.

4.1 Introduction

Defensics is a generation-based fuzzer. As discussed earlier, this means that test case generation during fuzzing is based on specifications. In Defensics, specifications are the basis for test suites, which include a collection of test cases for certain specification. Most of the suites are for network protocols, but there are also suites for different file formats, for instance. This thesis is only concerned about protocol test suites.

Each test suite needs to be configured before executing a fuzz campaign. Defensics uses its own file format as configuration file, called a test plan. Test plans are created from a test suite. They contain run specific parameters, such as target host and instrumentation method. Besides test plans it is also possible to run pure test suites from command line. Running pure test suites might require more parameters and is not the official recommendation, but they do not require a separate file. In addition, executing certain features from command line is not possible using a test plan.

Defensics is designed for black box testing, but it is possible to use grey and white box techniques for evaluation purposes [98]. For example, user can provide custom scripts that examine SUT and inform Defensics if certain criteria are not met. Defensics is also a stateful protocol fuzzer. This state-aware approach allows Defensics to analyze not only protocol messages, but also message sequences and exchanges.

The high-level testing setup consists of Defensics Monitor, test suite and SUT. Test suite is considered as a single network component with its own network configuration, including IP address. Test suite sends test cases to SUT and in many cases evaluates the results. Exceptions where Defensics Monitor and SUT communicate directly are SNMP-based

evaluation methods, for example. There can exist multiple parallel test suites that test a single system or multiple systems. However, for accurate root cause analysis of a fault, it is recommended to only run a single test suite at a time. Defensics recommendation is also to reset SUT between test suite executions to ensure SUT is in a healthy state before starting a fuzz campaign [99].

For automation purposes, Defensics offers command line interface (CLI) and HTTP API. For this work it was decided to use only CLI. Listed below are the reasons why HTTP API was not chosen:

- CLI makes more sense, because the CI workflow requires transferring files to scanner machine anyway. Sending the files and additionally making HTTP requests from CI server would seem more complex.
- Fetching results, especially those returned by interoperability probe, would require separate parsing methods when using HTTP API.
- When Using HTTP API, suites can be started sequentially or all in parallel. Starting only specific suites in parallel requires sending multiple POST requests.
- HTTP requests are messier than CLI commands.
- If Defensics server is restarted, new authentication token is created which needs to be fetched manually or extra effort must be used for automation.

The subchapters below describe features and configurable settings of Defensics. Because HTTP API is not used in this work, it won't be discussed further either. Hence, below subchapters concern only GUI and CLI execution.

4.2 Installation and licensing

The Defensics application, called Defensics Monitor, must be downloaded from an official download page. In order to access the official download page, user must have appropriate credentials. These credentials can only be acquired from Defensics support and they are linked to one user. From the download page it is possible to download either a Linux shell script or a Windows executable. Both shell script and Windows executable can be executed either via GUI or command line. Once the installation has finished, the Monitor can be started from desktop menu or from command line on both platforms.

All Defensics test suites require a license key. There are 3 different license configuration options, but the company recommended option is using a remote server. This remote server contains license keys for all available suites. The connection is made from Defensics GUI or from command line by setting the server's IP address and TCP port number. Once connection has been successfully established, test suites can be installed and therefore fuzzing can be started.

4.3 Configuration

There are multiple ways to configure Defensics. Defensics GUI has 8 tabs for different settings. The first 5 are settings for fuzz campaign pre-execution, and the last 2 are for post-execution. The remaining tab is for controlling an ongoing fuzz campaign. Settings there make it possible to start, stop and pause current execution, but do not affect the fuzzing otherwise. Hence, it will not be discussed further here. Sections below describe settings before and after execution, i.e. what possibilities there are to control fuzz campaigns.

4.3.1 Pre-execution

Basic

Basic settings vary a lot between test suites. The main idea stays the same however: provide parameters for connecting to SUT. At the simplest case this means only setting the IP address of SUT. In more complex cases basic settings can require port numbers, MAC addresses, cryptographic keys etc. Without valid basic settings connecting to SUT and therefore fuzzing is not possible. If basic settings have been configured incorrectly, interoperability tests will fail too.

One of the configurable settings is virtual interfaces. Defensics uses virtual MAC and IP addresses to disallow protocol messages being handled by host operating system's network stack. Both virtual addresses need to be unique in the subnet they are used in. If not specified by user, MAC and IP addresses are automatically chosen. Automatic choosing creates a problem in our cloud environment where allowed address pairs are limited for a VM port. In other words, only fixed address pairs are allowed. IP address can be set to a subnet using CIDR notation, but for MAC address no corresponding option is available.

Interoperability

Due to Defensics having complete protocol specifications, it supports every possible configuration of a protocol, which might not be the case for SUT. Interoperability checks which configurations are supported by SUT. No fuzzing is performed at this stage as only the valid cases for different configurations are sent to SUT. Interoperability results affect which test cases are executed during fuzzing, because only the interoperable test cases are selected. After all, it does not make sense to test unsupported features.

Interoperability test can be launched from the command line too. However, unlike GUI, CLI does not allow selecting specific protocol features. Instead, all the interoperable features are selected for test case selection. In most cases this should not be a problem, because test coverage should be as large as possible while minimizing redundant cases.

Advanced

Advanced settings affect test runs, but they are not as critical as basic settings. Advanced settings are also more similar between test suites. The main settings are related to run control, logging, packet capturing and CVSS scoring. Run control allows modifying settings such as timeouts, delays, looping and stop criteria. Logging defines the level of logging, e.g. are only valid and failed cases be logged, or should everything be logged. Packet capturing makes it possible to save network traffic for debug purposes. Lastly, CVSS scoring can be used to assign CVSS scores for found vulnerabilities.

Perhaps the most interesting advanced settings are timeout for received messages, amount of send attempts and different stop criteria. Timeout for received messages determines how long a suite will wait for a response from SUT. It can be set to a static value, that will always be waited but by default the value is dynamic. Dynamic value is automatically adjusted based on SUT responses. It still uses a default value which is also the time that will be waited before sending a case, but there is no fixed response wait time. Amount of send attempts defines how many times a case will be attempted to send, before assigning a verdict. Stop criteria has different options for stopping the fuzz campaign, such as amount of failed cases or total run time.

Instrumentation

Instrumentation is the act of observing SUT behaviour during fuzzing, i.e. evaluating test case effect. Its goal is to perform a health check for the SUT and report any discrepancies from expected state. How the health check is performed depends on the test suite, SUT and chosen method. This is also where classifications such as CRASH criteria must be considered: failure is not always catastrophic resulting in a complete system failure, but failures can be more subtle too. For example, containers may go into a restart loop or memory may be exhausted.

In some cases instrumentation might simply mean sending a valid case to SUT to see if there is a response. If there is, SUT can be said to be healthy and fuzzing can continue. Otherwise the conclusion is that SUT has reached an erroneous state, possibly implying a crash. In case of instrumenting more subtle failures SUT must be first manually assessed to find relevant symptoms which indicate a failure in the system and can therefore be used as instrumentation methods. In these cases instrumentation is very case specific. For example, different protocols have different message options for valid case instrumentation. Valid case instrumentation and other instrumentation methods supported by Defensics are discussed below and listed in table 4.1 based on Defensics User Guide [98].

Before delving into different instrumentation methods though, it is worth mentioning that instrumentation can be synchronous or asynchronous. In synchronous instrumentation health checks are performed depending on the test case, e.g. after *n* test cases, where

Table 4.1. *Instrumentation methods*

Method	Requirements
Valid case	Test suite that supports request/response model.
Connection	Always enabled for applicable suites.
Protocol	Applicable suite.
External	User made custom scripts.
SNMP-based	SNMP agent on SUT and appropriate port(s) open.
Syslog	Syslog support on SUT and appropriate port open.
Agent	Agent Instrumentation Framework on SUT and appropriate port open.
SafeGuard	Applicable suite.
ISASecure	ISASecure solution customer.

n is a positive integer. On the other hand, asynchronous instrumentation performs health checks periodically, e.g. after *t* seconds, where *t* is a positive number. These two methods favor either accuracy or speed. In case of a failure, detecting the cause is easier with synchronous instrumentation especially if *n* is small. This is because failure is often caused by a single test case rather than a combination of test cases [98], which makes locating the test case simple. The advantage of asynchronous instrumentation comes from less frequent instrumentation, which reduces fuzzing time but also makes fault detection harder.

Valid case instrumentation checks that SUT responds in an expected manner to a valid message. If the response is unexpected or there is no response within a predefined time frame, the test case is reported as a failure. Valid case instrumentation works synchronously and is executed after each test case by default. Even if a test case is reported as a failure, Defensics will continue using valid case instrumentation to get an expected response before sending a new test case. The amount of attempts, timeouts and other related parameters are configurable in advanced settings or instrumentation settings.

Connection-based instrumentation checks network connection between a test suite and SUT. It is enabled and cannot be disabled for all protocols that send a verification when connection has been established. For example, all TCP-based protocols work this way. It is noteworthy that any intermediate devices may cause false negatives or false positives when only using connection-based instrumentation. For example, if SUT crashes but a proxy between test suite and SUT is still opening a TCP socket, testing continues normally.

Protocol instrumentation works in a similar way as connection-based instrumentation. Instrumentation is done using request - response approach, but the requests are protocol dependent. Upon receiving a response its semantics are analyzed, e.g. what return code

is received. In order to assign a verdict, successful semantics must be predefined. For example, HTTP response codes starting with 1, 2 or 3 are acceptable, otherwise test case is marked as failed.

External instrumentation allows usage of custom instrumentation methods. Basically this means custom scripts that are executed at given point. For example scripts can be executed before test cases, after test cases or after the complete test run. External instrumentation allows user to freely determine how instrumentation is executed, but it also requires extra work on building the scripts. Examples of external instrumentation include monitoring of resources, log files and/or processes. External instrumentation is always synchronous.

Simple Network Management Protocol (SNMP) provides a way to reference data located in a remote system [100]. This data can be, for instance, CPU and memory usage data. To fetch specific data, user needs to know the corresponding object identifier (OID). Defensics supports SNMP Trap and SNMP Query instrumentation. The former works by SUT sending information to testing system once certain criteria are met, while the latter works by periodically querying status of the SUT. For SNMP instrumentation to work, the SUT must run an SNMP agent. SNMP Trap instrumentation sends data, including OIDs, that Defensics parses and analyses. For SNMP Query instrumentation, all necessary OIDs need to be specified. Defensics is also able to automatically fetch all the OIDs supported by SUT by using SNMP Scanner.

Syslog instrumentation is similar to SNMP Trap instrumentation. Whenever Syslog enabled SUT observes an event that is worth a syslog message, the message is sent to Defensics. Defensics then decides if the received syslog message is worth a failure. By default all syslog messages cause a failure. Syslog messages contain numeric severity and facility values, that can be used for filtering messages.

Agent instrumentation uses pieces of software specialised for a certain task, a.k.a. agents, on the SUT to send information to Defensics when an interesting event occurs. For the instrumentation to work, Defensics Agent Instrumentation Framework must be installed on the SUT. The framework has built-in agents that can monitor file or process events, for example. However, it also possible to create custom agents, that allow user freely to decide what is monitored.

Stateful design of Defensics allows it to analyze even complex protocol message sequences. SafeGuard instrumentation is used to detect security vulnerabilities in the system by observing the messages. SafeGuard instrumentation is prone to false positives, which is why manual review is required in most cases. Only cases determined as critical are automatically marked as a failure. SafeGuard is not supported by all test suites, but it is enabled by default for those suites that support it. There are various SafeGuard vulnerability checks that can be used. To name one example, SafeGuard instrumentation can

detect Heartbleed bug [91] by sending a Heartbeat request.

Lastly, ISASecure instrumentation monitors analog and digital waveforms to ensure they fit in a predefined criteria. For ISASecure instrumentation to work, customer needs to be part of Defensics ISASecure solution customers.

Test cases

Due to Defensics being a generation-based fuzzer, test suites are aware of the correct protocol specification format. This also means that test cases, i.e. modified versions of the correct messages, are predefined. The amount of test cases varies a lot, but some test suites can contain millions of test cases while others have only few thousand test cases. Each suite has certain amount of unique test cases from which a subset is selected by interoperability test and possible manual selection. Only interoperable test cases are used by default. Defensics is also able to loop test cases, allowing infinite execution. Once launched, infinite execution must be manually stopped. Of course the execution is not infinite in practice, as memory will run out at some point.

Defensics allows user to freely decide which cases are run and in which order. Test cases can be specified by test case index or by group name. Groups are a collection of similar test cases, e.g. cases that only alter a single field of a protocol message. Test cases are sorted by groups and by default selected cases are run in order. It is, however, possible to run the cases in a random order. Another option that does not execute test cases in order is balanced execution, which means selecting a meaningful subset from all test cases. For instance, instead of running first 1000 cases in order, which would only fuzz few message fields, these 1000 cases will be selected so that each field of the message is tested as much as possible. This way everything gets fuzzed, but not as much.

4.3.2 Post-execution

Results and Remediation

Defensics saves results of each fuzz campaign and interoperability test. The results contain multiple files that contain information about the campaign. For example, main log can save an entry for each test case, listing the contained anomaly and SUT response. Defensics GUI additionally makes it possible to view the exact test case messages and the anomalies contained in them in a graphical form. It is also possible to view runtime information, which shows how many test cases have been executed and how many are left, for instance. Result notes can be used for adding information about the test run or the tester. Users can freely decide what is written in notes, if anything. Results can be searched and filtered using notes later on.

Although the goal of Defensics is to find vulnerabilities, the ultimate goal behind fuzz

testing is to fix all found vulnerabilities. To do so, it is important to be aware of the cause behind an error, or vulnerability. Defensics makes it possible to rerun test cases that caused a failure. In the simplest scenario, only the case that caused a failure is rerun. However, sometimes a failure is caused by an earlier test case or by a combination of cases. Hence, user can define which test cases are rerun, e.g. 10 test cases leading up to the failure are rerun.

Remediation packages are a collection of files that can be used to reproduce a found flaw. The content is up to package creator. However, at least results that can be used to reproduce the fault should be included. Remediation packages should not contain whole fuzz campaign results leading up to a fault, but instead only the test case(s) that cause the fault. This makes reproducing the issue efficient. When importing a remediation package, the same test suite and its configurations can be loaded. The idea Defensics has behind remediation packages is that a tester can send the package to a developer, who can then easily reproduce the issue.

4.4 Current Challenges

Defensics has been part of the team's security testing for a while due to company requirements and recommendations. However, team members have not had time to thoroughly study Defensics. Hence, no clear definitions have been made regarding necessary test suites or their configurations. Instead, suites have been selected based on a tester's personal decision. These suites have been configured using certain settings that pass the testing easier than default settings. It would be beneficial to analyze which suites could actually be used and how they should be configured.

Another challenge with Defensics is speed, which is affected by a few factors. First, fuzz testing in general requires a large set of test cases. More cases there are, the more time it takes to complete the fuzz campaign. Secondly, Defensics does protocol fuzzing, where cases are send through a network which inherently takes some time. In addition, after instrumentation Defensics needs to wait for a response, which again goes through the network. Third, some cases take a lot longer than others. In the worst cases, total execution time is over 24 hours for a single test suite due to cases that take multiple seconds or even minutes. This challenge and its causes are discussed more thoroughly in chapter 5. Fourth, the SUT has multiple interfaces, which all need to be fuzzed. However, amount of parallel running test suites is limited. Defensics recommends not running more than 5 test suites simultaneously in GUI [101]. Lastly, operating the GUI takes time. For instance, loading test suites, configuring them and generating reports require notable manual effort.

When it comes to resource usage, the limiting factors of Defensics are memory, processors and storage. Different test suites require different amounts of memory but the official

recommendation is to have at least 2 GB of RAM for each suite. In addition, Defensics Monitor itself requires additional 2 GB of RAM [98]. So, for example, to run 5 suites simultaneously, there should be at least 12 GB of available memory. Test suites also need processing power. When running multiple suites simultaneously, one CPU core can handle multiple test suites. For optimal performance though, one processor core should run 1-2 test suites [101]. Defensics also recommends using at least Intel Core i5 processor. For disk storage, Defensics recommendation is to have at least 300 GB of available disk space for installation and log file storage [101].

Defensics GUI has additional recommendations, namely a graphics display adapter and sufficient amount of Java heap memory [101]. Defensics does not recommend running more than 5 suites simultaneously in the GUI, because results are temporarily stored in JAVA GUI components which causes large memory consumption [101]. Furthermore, the recommendation is to use command line execution over GUI for optimal performance. Real life test runs have shown that it is possible to run more than 5 suites simultaneously, but not without problems. Running multiple simultaneous suites makes GUI slow and in some cases the GUI has crashed, stopping all the running suites.

5. AUTOMATING DEFENSICS

This chapter describes the practical work that was done for the thesis. First, the selected Defensics suites are introduced. Then, all suite configurations that were chosen for CI are listed. The topic then moves onto general CI workflow, i.e., how does Defensics integrate into CI. Thirdly, it is described how Defensics Robot test suite and its test cases were formed. Lastly, the program created for running Defensics processes is examined.

5.1 Defensics configuration

The Defensics test suite configurations are described in this subchapter. First, all protocol suites that could be used to fuzz the given product are listed. From these suites a subset is selected for CI. The rest of the suites will not be used at least for this implementation. Reasons are given for excluded suites. Next is discussed which suite settings were modified from their default values. It is shown that certain situations cause significant delay when Defensics assigns a test case verdict and mitigation actions are given for these situations. As the last step, test case selection mode and a way to limit fuzzing run time is chosen.

5.1.1 Test suites

Defensics has numerous test suites, but only few of them are interoperable with the given product and environment. To filter which protocol test suites could theoretically be used, the full test suite list [102] was manually analyzed. Table 5.1 lists results of this analysis. All suites in the table could potentially be used to test the SUT. Suites that were excluded from the table are not interoperable with the SUT. In most cases this means that the protocol is not present in SUT. In some cases the protocol may be internally used by the product, but cannot be used from the outside and hence is not an attack vector.

Most protocol suites are for server-side testing, but Defensics also has suites for client side testing. These suites will not probe SUT but instead they will open a socket for listening connections from SUT. Fuzzing happens by sending responses to messages sent by SUT. This would naturally require adding functionality for repeatedly sending messages to Defensics. Because we want to keep security testing environment as similar to produc-

tion environment as possible, client side testing is not a compelling option. Also, client side testing is significantly more complex fuzzing method especially when executed during CI. It was therefore decided not use client side testing for CI, but rather do it manually if needed.

Table 5.1 lists suites that could be used for fuzzing the product, but not all of them are applicable for continuous testing. When the system test period is rather short and fuzz testing is only one part of security testing, it makes sense to prioritize some suites and exclude others. As discussed earlier, fuzz testing can be time-consuming with some fuzz campaigns lasting over 24 hours. However, excluding too much must be avoided to cover as much attack surface as possible. Suites in table 5.1 were further analyzed by considering their importance and executing test runs to gather final suitable test suites in table 5.2.

Table 5.1. *Possible test suites*

Test suite	Brief
X.509	Servers using TLS or other cryptographic protocol that utilizes X.509 PKI standard.
SSHv2 Server	Servers supporting Secure Shell 2.0, secure communications protocol.
TLS Server 1.2/1.3	Servers using Transport Layer Security (version 1.2 or 1.3), current de facto protocol for securing web traffic.
HTTP Server	Servers using Hypertext Transfer Protocol, application level protocol for transmitting hypermedia in the web.
TCP Server for IPv4	Servers using Transmission Control Protocol, transport layer protocol for IP networks.
TCP Server for IPv6	Same as TCP Server for IPv4, but for IPv6 implementations.
IPv4	Servers using Internet Protocol version 4, network layer protocol for networks.
IPv6	Same as IPv4, but for IPv6 implementations.
ARP Server	Servers using Address Resolution Protocol, network layer protocol for resolving MAC address.
ICMPv4	Servers using Internet Control Message Protocol for IPv4, network layer protocol for sending mostly diagnostic messages.
ICMPv6	Same as ICMPv4, but for IPv6 implementations.
Ethernet	Servers using Ethernet protocol, data link layer protocol.
Traffic Capture Fuzzer	Mutation-based suite for protocols.

The product runs OpenSSH server process, which means it supports SSH connections. SSHv2 Server test suite is therefore interoperable with SUT and could be used during fuzzing. There are, however, 2 main problems with the suite. First, although interoperable

Table 5.2. *Selected suites*

Test suite	Applicable Product Role
TLS Server 1.2/1.3	A
HTTP Server	A
TCP Server for IPv4	A + B
TCP Server for IPv6	A + B
IPv4	A + B
IPv6	A + B
Ethernet	A + B

with SUT, the primary focus of the testing should be on the product itself. In this case the focus would be on OpenSSH. Secondly, performance of the suite is poor. This is because the SSH protocol is complex. When running 1000 test cases in balanced execution mode, the average case execution time was 0.18 seconds. For reference, the time was 0.05 seconds for HTTP suite and 0.02 seconds for TCP for IPv4 suite. The execution time still seems small, but when running hundreds of thousands of test cases the difference becomes more significant. For example, the SSH suite contains approximately 700,000 cases. If the cases on average take 0.18 seconds, this adds up to 35 hours. For HTTP suite this time is 9.7 hours and for TCP suite it is 3.9 hours. For these reasons SSHv2 suite was decided to be excluded from CI testing.

ARP Server test suite is for testing ARP protocol implementations. ARP is used to resolve MAC address from corresponding IP address. ARP is implemented in the operating system of SUT, for example in Linux kernel. User's should not directly interact with ARP implementation, but rather it acts as a service for other protocols [103]. ICMPv4 and ICMPv6 test suites are for testing ICMP protocol implementations. ICMP is used for sending information about error conditions and other diagnostics. Like ARP, ICMP is implemented in the operating system and it should not be directly accessed by user because ICMP messages are sent via other protocols [104]. Like SSHv2 suite, ARP and ICMP suites are not focused on testing the product. Additionally, there are already test suites for data link layer and network layer. Hence, ARP and ICMP suites were excluded from CI testing.

HTTPS, which uses TLS, is used in product role A. TLS needs public key certificates and the certificate is in X.509 format. Therefore Defensics's X.509 test suite could be used for fuzzing. The suite, however, operates differently than other discussed suites. Instead of directly sending test cases to SUT, the test cases, i.e. invalid certificates, are only generated and sending is left as user responsibility. Defensics does offer options for automatic sending, but it still needs user made scripts and/or configurations. One option would also be running TLS suite simultaneously by looping valid TLS message

and feeding a fuzzed certificate as part of the message. Either way, X.509 suite would require special effort and since it also does not directly test the product, it was left out of CI testing for now.

Unlike almost all other test suites, Traffic Capture Fuzzer (TCF) is mutation-based and not generation-based. As the name suggests, the suite generates test cases by mutating traffic capture files. There is no live traffic interference though, but instead a single capture file is imported. Traffic must be captured using an external tool such as Wireshark. Because there are no proprietary protocols in use and there already exists a suite for each used protocol, TCF does not add much value. Even the official recommendation is to use a fitting suite if one exists. TCF will therefore not be used in CI testing.

5.1.2 Test suite configuration

As shown in chapter 4, Defensics suites can be configured in multiple ways. Most default settings are already sufficient for testing the product and can therefore be left unchanged. There exists a few settings that should be changed for CI testing though. First, basic settings must be changed for interoperability test. Secondly, advanced run control settings provide useful parameters for improving run efficiency. Same goes for instrumentation settings. Lastly, test case settings allow changing test case selection mode. They also make it possible to select only a subset of cases for execution.

Test suites contain thousands, hundreds of thousands or even millions of test cases. When these cases are sent through a network, the fuzzing process will inherently be time-consuming. But even if test suite execution is slow, it is still time-wise manageable especially if parallel execution is used. However, time-wise real problems arise when SUT stops processing messages, often due to high load caused by fuzzing. In these cases Defensics may mark test cases as skipped or failed. Whether case is marked as skipped or failed depends on instrumentation. Successful instrumentation yields a skip, while unsuccessful instrumentation fails the case. Note that test cases might be marked skipped in other cases too. For example, when fuzzing with uninteroperable cases. In such situations concluding the verdict could be just as efficient as with successful cases.

By default, test cases will be attempted to be sent 3 times and instrumentation will be done only once. If there is no response to any test case within send attempt limit case is marked as skipped or failed. A failed case does not stop fuzzing, but instead instrumentation will continue until a response is received. Furthermore, delay between instrumentation attempts will progressively increase until the maximum time is reached, which is 12 seconds by default. Even if test case is only skipped, the test suite could still wait default timeout of 1 second for a response. Since there is 3 attempts by default, each skipped case will take approximately 3 seconds. This is huge difference compared to successful average execution time of a test case, which in the case of TCP for IPv4 suite

was only 0.02 seconds. Furthermore, instrumentation time is still added on top of this. What is worse is that previous fuzz runs have shown that usually SUT stops responding to several messages instead of just one. In other words, if a case gets skipped, many subsequent cases will also be skipped.

To address the performance problems caused by unresponsiveness of SUT, suites must be configured in a way that minimizes time spent on analyzing such cases. First, maximum fail amount is set to 0, meaning that suite will stop if a fail occurs. This makes sense because manually executed runs have shown that fails occur rarely and they should anyway be manually analyzed despite the amount. Furthermore, a suite does not get stuck on failed cases and slow the whole pipeline this way. Secondly, because normal test case execution time for all selected suites is couple hundreds of a second at most, it should be safe to lower the default timeout of 1 second. However, we do not want to set this too strict in case of random network delays. For the same reason there should be at least a couple send attempts. Therefore, dynamic timeout was set to 300 ms and amount of send attempts was kept at default. Instrumentation fail limit was increased from 1 to 5 to further add some leeway. After all, suite stops if at least 1 fail is encountered. Increasing instrumentation frequency would increase fuzzing speed, since instrumentation is done less often. It was, however, decided that accuracy should not be reduced and frequency was kept at 1.

The instrumentation methods themselves were kept simple. SNMP-based, syslog and agent framework instrumentation would all require extra SUT configurations and hence were chosen not to be used. ISASecure is not applicable because there is no ISASecure customer subscription. Valid, connection, protocol and SafeGuard instrumentation are used if they are set on by default. Also instrumentation messages were kept as default. External instrumentation could turn out to be useful because it can be used to check SUT internals, such as container status or logs. However, doing this after every test case would be inefficient. It would make more sense to do such instrumentation less often, although then accuracy is worse. External instrumentation was, however, not implemented for this thesis work and it was instead left as future work.

Although ideally all interoperable tests would be run, sometimes running a subset of cases is required to stay within time constraints. Hence, limiting test case amount should be an option. Defensics has different options for this, namely maximum case amount, maximum run time and trim percentage. Trim percentage allows selecting a percentage of cases from all cases, but can only be used with specific test case selection modes: trim and random. Trim executes cases sequentially as done by default, while random mode selects cases in a random order. Random execution is based on a customizable seed value, which is 0 by default.

Third usable test case selecion mode is balanced. Balanced mode is like random execu-

tion, but instead of choosing cases based on a random seed, Defensics chooses cases equally among test groups. The objective of balanced execution is to choose a meaningful subset of cases from all cases. This guarantees usage of all test groups, which in most cases implies fuzzing all fields of a protocol message. Drawback of balanced mode is that trim percentage cannot be used with it. Therefore, to limit test cases with balanced mode, another limit must be used. Maximum run time limit was chosen for this purpose. When suite reaches the limit, the fuzz campaign is stopped despite how many test cases have been executed. Maximum run time can also be used to mitigate prolonged run times caused by unresponsive SUT. Regarding CI, the time limit of the particular job needs to be taken into account, since all needed suites must be completed within the limit. Maximum run time was set only to 1 hour to guarantee reasonable execution time even with limited target machines and few suites running in parallel. Running most or all suites in parallel against multiple identical target machines was left as future work.

5.2 Executing Defensics in CI

Out of the two security testing environments presented in chapter 2, cloud environment felt like the natural choice for CI purposes. This is due to flexibility of virtual machines. Scanners, targets and possible other machines can be easily created and destroyed. Resource assignment, e.g. setting enough memory and virtual CPUs is trivial. Furthermore, these environment changes can be made automatically. For example, it is possible for CI pipeline to execute jobs that first create all target VMs, then perform scans against them and finally destroy them. Similarly, scanner VMs can be created during pipeline execution.

Another great advantage that the cloud environment offers is parallelization. Because Defensics scans can take a long time to finish, it can be a good idea to run a single fuzz campaign against multiple identical targets. For example, if a test suite has 1 million test cases, they can be split to 10 subsets of 100,000 cases and run against 10 identical targets. This offers opportunity for great time savings. It is theoretically possible to miss a specific combination of test cases leading to a failure by dividing test cases into subsets, but probability of this can be considered low. After all, failures are usually caused by individual test cases, as discussed in earlier chapters. Besides splitting cases of individual test suites, different interfaces can be split among identical targets. For example, each target could be used to fuzz one TCP port.

The CI workflow was explained in chapter 2 and visualized in figure 2.6. For this work, a stage was added to the security pipeline. The stage has a job that executes a Defensics Robot test suite, which was also created for this work. Two Python modules were also created: one for implementing business logic for the Robot suite and one for executing Defensics CLI commands on a scanner machine. The former has methods for performing

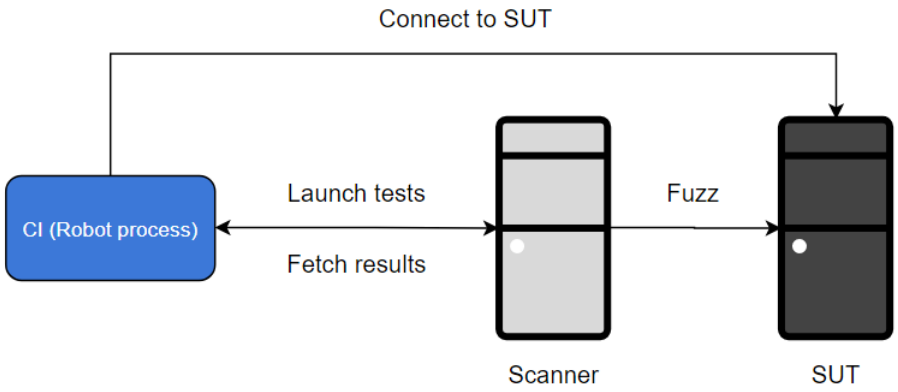

Figure 5.1. *Defensics as part of CI*

remote operations via SSH, such as transferring files, creating directories and launching scans. The latter will be explained in detail below.

When specifically considering Defensics and CI, the interesting part is what happens when the security pipeline is started. This is visualized on a high-level in figure 5.1. The connection between CI server and SUT allows preconfiguration etc. Currently this connection is only used for product installation. On a lower level, before fuzzing the product is first installed on the target machine and its condition is verified. These steps are related to the overall CI structure and not to strictly fuzzing. Unlike fuzzing job, they existed already before this work. After verifying successful installation, the target is ready for fuzzing. The CI fetches Robot test suite, the code for running Defensics and configuration files (see below) from testing code repository. Upon Robot suite initialization, the code and configuration file corresponding to the Robot test case is sent to scanner machine. After initialization, a command is sent to the scanner that executes the code using the provided files and parameters. Once fuzz campaigns have been completed, result files are fetched from scanner machine to CI server. Job verdict is then assigned and sent for observation.

Assigning a verdict is a key aspect of testing. Because Defensics does not use sophisticated white box techniques and instrumentation was chosen to be simple, amount of false positives is expected to be low. For this reason it is reasonable for a job to fail whenever any of the Defensics scans fail. Defensics CLI allows rerunning failed test runs, which could be implemented as a feature. However, failures are not always caused by individual test cases and sometimes system might be considered robust enough, even if a test fails. To recall FTMM from chapter 3, transient failures happen when failures cannot be easily reproduced and they can be considered acceptable. For this reason and the requirement from chapter 2, all failures are manually analysed and manually given a verdict afterwards.

As the core aspect of this thesis, a program was created for running Defensics suites. The program is called DefensicsPy and it was decided to run using pure suites instead of running commands with test plan files. Upon running DefensicsPy, the first step is checking for any absent or outdated suites. If so, all of these are downloaded and installed automatically. This ensures that scans are always executed using the latest test suite versions. After updating suites, the subnet is scanned for available IP addresses. The addresses are assigned for each test suite to be run. Thirdly, the test suites, i.e. scan processes, are started. The scans continue until all interoperable cases are run, time limit is reached or when a failure occurs. As a final step, reports are renamed to match the suite and interface, and then moved to another directory on the scanner machine. Further details are provided below.

5.3 Robot test cases

To not mix Robot test cases with Defensics fuzzing test cases, Robot test cases are called just Robot tests in this subchapter. Robot tests are the highest abstraction level when it comes to the code written. They use Robot keywords to call Python methods, which connect to a scanner machine and start the actual testing. The Robot calls made in CI or by a user specify which Robot tests are executed. Tests can be called using their names or tags assigned to them. Tags can be used to select tests without specifying test names. This is done by specifying the Robot test suite (in this case Defensics) and then including and/or excluding tags. When there are many Robot tests and/or their names are long, tags usually allow simpler commands.

```
1  DEFENSICS_PROUCT_ROLE_A
2      [Documentation]  Scan product role A.
3      [Tags]  role-a
4      ${scan_verdict} =  Launch Scan  ${CFG_ROLE_A}  role-a
5      Fetch Results  role-a
6      Should Be True  ${scan_verdict}  msg=SCAN FAILED!
7
8  DEFENSICS_PROUCT_ROLE_B
9      [Documentation]  Scan product role B.
10     [Tags]  role-b
11     ${scan_verdict} =  Launch Scan  ${CFG_ROLE_B}  role-b
12     Fetch Results  role-b
13     Should Be True  ${scan_verdict}  msg=SCAN FAILED!
```

Program 5.1. *Robot test cases for Defensics Robot test suite*

Robot tests were created for each target. The tests and their tags were named to differentiate product roles. The Defensics Robot test suite imports modules, which consist of

all the necessary methods for executing the tests. When a Robot test suite is called, first all necessary SSH connections are opened, i.e., connections to scanner and SUT. If successful, the tests and the scanner machine are initialized. For test initialization, temporary directory is created on the scanner machine, which will include DefensicsPy, configuration file and finally the results. These, besides results, are transferred next. Once initialization has been completed, the tests will be run. Robot tests select corresponding configuration file and execute a Python method, that executes the scanning script in the scanner machine and waits for scans to finish. When the scanning has been fully completed, results are fetched to a local result directory and scan verdict is given based on the return value. The verdict will also be given for the job in CI pipeline, e.g. a failed scan verdict will fail both Robot test and the job in CI pipeline.

Program 5.1 is a simplified version of the Robot tests created. There are 3 keywords in use: Launch Scan, Fetch Results and Should Be True. The first two were created for the Defensics Robot test suite, while the last one is a global keyword. Launch Scan passes the configuration file name for the Python module, which connects to the scanner machine, and returns the verdict upon scan finishing. Fetch Results calls the Python method which will retrieve all results that were gathered. This will happen regardless of scan verdict. Finally, the Robot test will either pass or fail depending whether scan verdict is true or false. If it is false, a message will displayed to state that.

5.4 Code implementation

As stated earlier in chapter 4, Defensics CLI was chosen for the automation. Defensics CLI has almost the same functionality as GUI with only few unnecessary functionalities missing. The CLI commands are launched from a Python program, that allows a controlled parallel execution of suites. Parameters for the commands are mostly given in a configuration file, with a couple hard coded parameters. Basic settings and other settings that should be configurable for each run are given in a configuration file. The file contents are analyzed more thoroughly below. The hard coded parameters are for interoperability probe and post-run report creation. They are applicable for all test suites and should remain constant between runs.

Because the main objective was to create code for CI, configuring must be suitable for it. The CI fetches code from testing code repository and executes predefined Robot commands in release testing repository. The code does not only include the business logic for running Defensics commands but also the needed configuration file. There is no user to modify configuration files or command line parameters at this point. Changes to configuration files or to business logic must made by updating code in testing code repository, while changes to Robot commands are done in release testing repository. However, there may be need to run the scripts without CI, for example to rerun certain

test cases. For these occasions configuration and test cases were made user friendly.

Configuration files exist to provide an easy way for changing run specific settings. In theory, there could be only one configuration file that would be modified before each run. However, due to large amount of target interfaces and needed parameters, this would require a large amount of Robot test cases, command line parameters or a long configuration file. Neither of those options are very practical or user friendly. Therefore, each target (product role) has a separate configuration file. This is a practical choice, because there are only 2 separate targets and interfaces between them differ quite a lot. There also exists a separate test case for each target, which automatically chooses the correct configuration file.

Configuration files use YAML syntax. Defensics test suites are lists of dictionaries, that have one or more targets as values. Targets include basic settings for given test suite, e.g. target IP address, port or URI. Because configuration files are specific for certain product role, all targets defined should be of the same role. It is possible to give settings for different product roles, but for example report directories will be misleading in this case. Besides test suites, other settings are given as simple dictionary mappings. These settings include everything that was considered nice to have configurable: path to Defensics application on the scanner machine, subnet CIDR, test case selection mode and test case indexes to name a few.

For running Defensics and wrapping the CLI commands, a program named DefensicsPy was created. DefensicsPy launches Defensics processes using Defensics CLI commands. Parameters for the commands are given in a configuration file or they are hard coded, as described above. DefensicsPy uses several classes, which are shown in UML class diagram in figure 5.2. The diagram is conceptual, i.e., it does not contain class methods or variables because the main focus is on higher level implementation along with suite configurations instead of implementation details. Command line arguments for DefensicsPy are path to configuration file, path to suite directory and path to result directory. Both configuration file sending and result directory creation are done during Robot test initialization. Suite directory should already exist on the executing machine. This is actually guaranteed by DefensicsPy, because it handles suite download and installation too (see below sections). Once suite execution has started, logs and run time results are generated to the result directory already during fuzzing.

DefensicsRunner is the main class responsible for controlling the execution flow of DefensicsPy. Only command line argument parsing is done before starting the runner object. DefensicsRunner uses Pinger to find available IP addresses, SuiteController to update suites and finally launches one or many DefensicsProcess objects. ConcreteSuite is a generalized term made for figure 5.2, as listing all suites in table 5.2 would require too much space. The concrete suite classes are implemented based on the abstract Suite

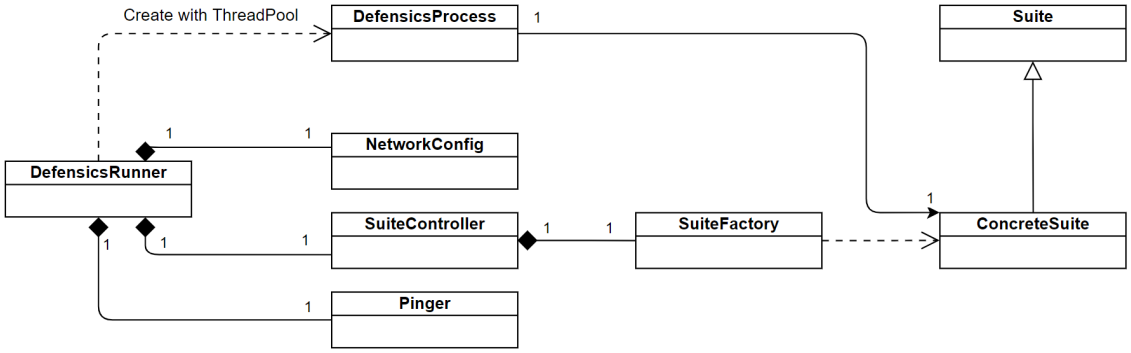

Figure 5.2. *DefensicsPy UML Class Diagram*

class. ConcreteSuite instance is fetched via SuiteController and SuiteFactory and then given as a parameter for each created DefensicsProcess. NetworkConfig, a class for configuring network related parameters, is created after finding available IP addresses. An IP address along with other network related parameters required by a specific Defensics suite are also given as a single parameter for DefensicsProcess instances.

Pinger class was created because Defensics suites need their own unique IP address. Defensics has option for automatic network configuration too, but due to the allowed address pair restriction in cloud environment (see chapter 4) it was decided not to be used to ensure correct configuration. To assign a unique IP address for a test suite, the free IP addresses in the subnet need to be found. This is done by scanning the subnet provided in configuration file using ping command and marking the hosts free that do not send a response. Because sequentially pinging even a relatively small subnet is slow, Python's multiprocessing module is used, making the scanning significantly faster. Also timeout options of the ping call are shortened for faster verdicts. Once available addresses have been found, test suites try to reserve one for themselves. If there are no available addresses, the test suites will have to wait for one to free up. Otherwise they are ready to be launched. After a test suite is finished, it will free the address pair it used. This pair can then be reused by a new test suite. In addition to subnet CIDR, addresses included in the cloud's DHCP pool are given in configuration file so they can be excluded.

For each suite, i.e., fuzz campaign, a new DefensicsProcess object is created. The class objects launch a subprocess, which is a process for the executed test suite. The class object stores the executed command along with a process ID, used address pair etc. The address pair is retrieved using a Pinger class object and the pair is freed when the process finishes. The pair can then be used by possible new processes waiting in queue. Processes are launched using multiprocessing module's pool.ThreadPool class which takes care of the queue along with other multiprocessing details. It also determines the amount of suites executed in parallel. To be more precise, the amount is determined by CPU core amount that is given in a configuration file.

The full scan command used in DefensicsProcess consists of different parts. For example, base command, interoperability parameter and suite options are different parts. Each part is built sequentially, and finally concatenated at the end just before execution. Test suites have different basic setting and network parameter structure which is why commands need to be created based on the given suite. The rest of the command is built similarly for all suites. Each suite class object also stores the official suite name (without version number) to locate correct suite and its latest version from given suite directory. This implies that suite directories are expected to use default naming, for example "tcps-8.5.0".

If a configuration file contains an interface which cannot be reached (e.g. a closed TCP port), the interoperability test will fail. In this case execution will continue and a warning is logged indicating which suite and interface was skipped. Defensics does not have a specific return code to identify a failed interoperability test from a real failure. Therefore, process output is used to determine this scenario if return code indicates anything but a successful scenario. But interoperability test might also fail because SUT is experiencing too much load and refuses to handle new messages. For this reason DefensicsPy will run processes only in pools the size of parallel suite count given in configuration file and recover for certain amount of time before launching another pool. Recovery time was set to be only 1 minute by default to not prolong the run too much. It is possible that the recovery time is not enough in some cases, which is why it is recommended to use multiple identical targets instead of just one SUT.

Once interoperability test has passed, suite will either successfully complete or it will fail. Nothing will be skipped anymore. In a successful scenario, HTML report is created for proof that the interface has passed fuzzing using the given suite. On the other hand, if a suite fails, DefensicsPy will throw an exception and terminate itself. Termination will happen after all processes in the pool have finished. This will return an erroneous return code, which CI server can use to detect a failure. The failed suite and the interface will be logged. In addition, a summary file is created to indicate which case caused the failure. This summary can then be fetched from the CI machine, as the information in the summary should be enough for debugging purposes. If further details are needed, all files generated by Defensics are located on the scanner machine, including remediation files for failed suites.

Occasionally a new updated version of a test suite is released. Defensics CLI provides command for listing all suites not present on the machine, including new versions of existing suites. By defining which suites are used in DefensicsPy, the suites can be filtered from the command output. After that, if there exists any suites to update, they are individually updated using a separate command. If there is nothing to update, execution will continue normally. If an update fails, it will be skipped and DefensicsPy resorts to older suite version. If there is not even an older version installed, suite will fail. Checking,

downloading and installing of suites is handled by SuiteController class.

6. ANALYSIS AND DISCUSSION

In this chapter the work done in chapter 5 is analyzed and discussed. This chapter is divided into 4 subchapters: execution time comparison, comparison to manual workflow, limitations and future work. First, execution times of the new suite configuration are compared to execution times of the previous configurations by executing test runs using each of the configurations. Then, new automated fuzzing workflow is compared to manual workflow by analyzing how tasks related to Defensics were done and how they are done now. Third subchapter lists and analyzes limitations of the work. In the final subchapter it is discussed what improvements could be done in the future.

6.1 Execution time comparison

Suite configurations were modified for CI but their possible impact on fuzzing were not analyzed. To analyze the impact, a test setup must be created. Creating a good test setup is hard because the made configurations have effect only when SUT stops responding to fuzzed messages. Setting SUT in such a state would require either modifying its internal structure or generating heavy load like in a real fuzzing scenario. Because the product is treated as black box in this work and inspecting its internals would require excessive studying, only the latter is a viable option.

For analyzing modifications to suite configuration, 2 realistic setups that stress SUT with heavy load were created. In the first setup 10 suites compatible with product role A were selected for fuzzing. These 10 suites are run in parallel using the code created for this work. Settings for the suites, however, are changed to match 3 different configurations: default, old and new. Default settings are those set by Defensics. Old settings match settings used previously when the team has run Defensics manually. New settings are those made for this work, except that maximum fail limit and run time were not set to make total run time comparable. The second test setup is almost the same as the first one. The differences are that only TCP Server for IPv4 suites are used, there are 15 suites instead of 10 and testing is done for product role B. SUT is rebooted after every test run.

The results of the first setup are in table 6.1. The table first lists suite settings for each configuration. Timeout is the time until turn changes to next case in queue. Send attempts define how many times a single case is attempted to send to SUT. Fail limit is short for

instrumentation fail limit, i.e., how many times instrumentation is attempted. After the settings, the table lists the total amount of skipped and failed cases observed during fuzzing. The last column indicates total execution time for running all suites in parallel. In practice this is the execution time of the slowest fuzz campaign plus time used for Robot test suite initialization etc. Because same suites were used agains the same SUT, total test case amount is same for all configurations. This amount was approximately 8 million cases.

Table 6.1. *Suite configuration comparison for mixed suites*

Config	Timeout (ms)	Send attempts	Fail limit	Skipped	Failed	Total time
Default	1000	3	1	0	306	10 h 34 min
Old	1000	10	10	0	0	10 h 32 min
New	300	3	5	1	0	10 h 36 min

From the total execution times we can see that there has not been any improvement when using the new configuration. In fact, the old configuration seems to be the fastest. However, when comparing differences to total execution times, the differences become negligible. Hence, it can be said that fuzzing takes same amount of time with all configurations. This result is expected because the settings should affect execution time only if SUT becomes unresponsive and results in many skipped and failed cases. In this test setup there were only few skipped and failed cases. Furthermore, if skips or failures would occur in suites that take less time than other suites in successful scenarios, the differences are hidden.

Indeed, detailed analysis showed that all the 306 failures were observed in TCP suites which are not the slowest suites. In this case. however, the differences were only about 15 minutes between default and new configuration. The reason why default settings produce failures and others do not is the fail limit. When dealing with heavy load, SUT is prone to leave a few messages unanswered within the time limits. When fail limit is higher, it becomes less likely that SUT does not send a response to at least one instrumentation attempt. The fail limit is most likely also the reason for 0 skipped cases with default settings. To recall from chapter 5, skipped cases are assigned only if instrumentation succeeds within fail limit, which was only 1 for default configuration. The one skipped case with new settings is assumed to be a random occurrence related to heavy load.

With the objective to cause SUT become unresponsive, second test setup is more demanding. There are 15 TCP suites running in parallel against a single SUT. The total test case amount adds up to 11 million. Configurations, i.e., suite settings are still the same as with previous test setup. Results of the second test setup are listed in table 6.2.

From the results in table 6.2, it is clear that SUT had harder time to deal with the heavier load. Each configuration had over a hundred thousand skipped cases and default config-

Table 6.2. *Suite configuration comparison for TCP suites*

Config	Timeout (ms)	Send attempts	Fail limit	Skipped	Failed	Total time
Default	1000	3	1	100116	18244	7 h 25 min
Old	1000	10	10	100002	0	6 h 55 min
New	300	3	5	108012	0	6 h 10 min

uration had almost twenty thousand failed cases. Failed cases with default configuration are again explained by the instrumentation fail limit. When comparing the execution times, it does look like the new configuration is slightly faster than the other two configurations. As numbers, new configuration is approximately 11% faster than old configuration and approximately 17% faster than default configuration. However, stating this would be inaccurate as the first test setup already shows. These results apply only for this particular test run. It is expected though that the new configuration is slightly faster whenever many cases get skipped. Interestingly vast majority of skipped cases with all configurations were caused by a single interface, as the generated reports show. This is an indication that it could be worthwhile to investigate the component behind the interface and see why it produces so many skipped cases compared to other components.

The data in the tables or reports does not tell everything. More conclusions can be made by looking at the logs generated by Defensics. For instance, from the logs it can be seen that most skipped cases do not use the full timeout. This is due to Defensics's dynamic timeout, i.e., timeout is adjusted during fuzzing to be as fast as possible. Dynamic timeout is the reason why total execution times are not significantly longer with a hundred thousand skipped cases. To be more precise: when individual send attempts take same time as passed cases that only need 1 attempt, the full time it takes to assign a verdict is approximately the product of send attempts and time of a passed case with 1 send attempt. For example, with old suite configuration skipped cases are handled 10 times slower than passed cases with a single send attempt. In other words, skipped cases slow the fuzzing even with dynamic timeout. For this reason and to avoid skipped cases caused by unresponsive SUT, multiple identical targets should be used instead of a single SUT.

6.2 Comparison to manual workflow

Launching suites happened previously from Defensics monitor, i.e., Defensics GUI. This workflow requires connecting to scanner machine via remote desktop application and then launching Defensics. Once GUI has opened, tester loads a few suites one-by-one. Target's IP address, suite's virtual addresses and other settings are then set for each loaded suite. Suites are renamed to differentiate them from each other and finally launched one-by-one. Due to Java heap memory restrictions, not all suites can be run si-

multaneously. Hence, the process must be repeated a couple times for running rest of the suites. Conversely, The CI workflow allows setting all required settings into configuration file, which must be done only once. After that, the settings do not need to be modified unless targets or testing environment are updated so that e.g. IP addresses change. Using the configuration file, CI starts suites automatically on scanner machine.

To comply with transparency requirements, reports must be created after Defensics scans are over. From GUI this is done from results tab, where name and other options must be set for each report. The process is done for each suite. After generating all reports, they must be transferred from scanner machine to other storage. When Defensics is run from CI, reports are automatically generated, named and moved to the executing server after test run. From there they still should be moved to some long term storage. Generation of testing documents based on reports also works the same way for both manual and automated execution.

Whenever there is a new version of a suite available, it will be detected by the created program. Assuming there are no installation errors, this means that fuzzing is done using latest suites. If done from GUI, tester needs to open suite browser and manually install each update. Likewise, the Defensics monitor can be updated from suite browser. Monitor will not be automatically updated in CI either. Instead, the scanner machine must be accessed and monitor updated using command line.

6.3 Limitations

As can be seen from execution times in tables 6.1 and 6.2 and from chapter 5, run times of Defensics in successful scenarios have not improved significantly. There is not enough target machines or scripts to automate their creation and/or maintenance either. This is required for executing all suites in parallel. Hence, full execution is still time-consuming and not practical for CI. Because of this, all test cases of the suites will not be run in CI for every release candidate but instead smaller subsets will be selected. These are determined by suite's maximum run time and order of Defensics's balanced execution. Full runs will still be done, but the triggering happens manually or on a weekly basis. A separate pipeline was created for this purpose. CI pipeline and virtual machines make it possible to automate environment modification, e.g. creating and destroying scanners and targets. However, currently no such functionality is in place. The machines must exist before a pipeline is run. If there are multiple target machines, it is also recommended to have static IP addresses. Otherwise configuration files need to be updated whenever machines are recreated. Furthermore, before installing the product into target machines, the operating system must be correctly configured, e.g. running hardening scripts. This configuration is not automated at the moment and therefore not part of the pipeline.

The scanner machines must be initialized before using them. Since code was created us-

ing Python and certain libraries are only supported from version 3.8 onwards, the scanner requires Python version 3.8 or higher. The configuration files also need PyYAML package for parsing. Regarding Defensics, the application itself and Java must be installed. As mentioned in chapter 4, when downloading Defensics a certain download page must be accessed using provided credentials, which are tied to one employee. Although it could be possible to create a download script using the credentials, it imposes a risk to the particular employee. If something malicious, intentional or accidental, is done using the credentials, the employee will be held responsible. A better option would be to have a specific account for automation, but unfortunately this is not possible. Additionally, all possible configurations related to previously mentioned requirements must be done.

Although suites are automatically downloaded and installed, this does not apply to Defensics itself, i.e., Defensics monitor. Defensics does not offer option for automatic installation. When there is a new monitor update, it must be manually downloaded and installed, including all configurations needed by installation script. Fortunately, this is minimal effort and there are only few updates per year. There is no automation for configuring licenses either. Unlike the monitor however, automating license configuration is possible using Defensics CLI.

Due to large amount of interfaces and therefore required fuzz campaigns, functionality for dividing test cases for a single test suite was not implemented. If there was only a couple of suites and interfaces, this feature would make more sense. But currently there are several of both suites and interfaces, which would require significant amount of identical SUT machines. Therefore it is only possible to split interfaces among different targets, e.g. different TCP ports are used to fuzz different targets.

Configuration files offer a simple syntax and a common place for Defensics parameters, but modifying them is laborous from CI perspective. As can be seen from figure 2.6, modifications cannot be made where CI is located. Instead, they must be made in testing code repository. It would be easier from CI perspective if parameters could be given as robot variables. However, the reason configuration files were made is that it should be easy for users to do custom scans too. Ideally parameters could be changed from Robot side while maintaining some easily readable syntax for humans. Having a need for doing such modifications should be rare though.

6.4 Future work

The natural next step for the work done would be to increase the level of automation. First, everything required by Defensics and DefensicsPy should be done automatically. Downloading and installing Java, Defensics monitor, licenses, Python and PyYAML could be done in a separate stage in the security pipeline. Successful installation of these would then be required by Defensics scanning stage. Once everything related to Defensics is

handled automatically, the focus can move to environment. Here the first step would be to integrate OS level configuration of target machines into CI. After that automatic creation of the machines could be considered. The security pipeline would then cover creation of target machines, configuring them, installation of the product, installing Defensics related dependencies and finally performing the scans. The created target machines could be destroyed as the final clean-up stage.

The other step is for suite configuration. In this work the configurations were analyzed but not optimized. In could be worthwhile to check more thoroughly how e.g. different timeout and instrumentation fail limit values affect execution time. Valid case instrumentation also uses default messages for instrumenting, e.g. TCP SYN and TCP Reset for TCP suites. It could be investigated if other message types are more efficient for instrumentation. Additionally, use of other instrumentation methods could be investigated. As discussed before, external instrumentation could provide useful data during fuzzing. However, monitoring SUT does not need to be responsibility of the fuzzer. It is also possible to use some separate tool for monitoring, although identifying possible failures becomes more difficult then.

One aspect that was not really considered in this work is limiting amount of executed test cases. For each selected suite, the full test case amount is executed by default. As explained in chapter 3 though, there is an infinite amount of cases that could be used. The cases in Defensics are selected by its developers by some criteria and we trust those cases are enough. No proof, however, exists to support this belief. It is possible that the current case amount is even too much, i.e., there is redundancy. If this is true, we could use a subset of cases with same results, while reducing execution time. In order to do so without compromising testing quality, one would need to prove that a certain subset is enough for (near) optimal fuzzing result. This is an idea to consider for future work.

7. CONCLUSIONS

This thesis was done for a team in Nokia, who have been running Defensics fuzzer as part of their product's security testing. The product consists of 2 product roles, which are treated as separate targets. The fuzzing has been done manually via Defenesics GUI. This approach has had 2 main challenges: speed and efficiency. Manual operation via GUI is time-consuming and running several test suites in parallel is not possible. The other problem is that the scans themselves take a lot of time. When combining these two challenges, it is evident that there should be a better option. Therefore, the objectives of this thesis were to integrate Defensics fuzz testing as part of CI and investigate if the fuzzing can be speed up. The scope was set to concern only protocol fuzzing with Defensics and the product was treated as a black box. Out of two security testing environments, bare metal and cloud, the cloud environment was chosen for automation.

Regarding the automation, there were 3 main components in play: CI, Robot framework and DefensicsPy. DefensicsPy is the program created for this work. It launches fuzzing processes using Defensics CLI and supports running multiple suites in parallel. Robot framework was used to create a Robot test suite for Defensics. Robot test cases were created for both product roles. A separate Python module was also created to hide business logic of the Robot suite. Defensics was added to CI by creating a stage and a job for it into existing security testing pipeline. To limit execution time of the pipeline, a subset of test cases were selected by limiting suite run time. Another pipeline was created for running all test cases. Both pipelines use DefensicsPy via the Defensics Robot test suite.

After analyzing Defensics behaviour, it was reasoned that Defensics acts the slowest when SUT becomes unresponsive and causes Defensics to assign skip or fail verdicts. A few parameters were then identified that could affect execution time. Two of these, maximum run time and maximum fail limit, were taken into use to stop execution if time limit is exceeded or any fails occur. The other parameters will not stop execution but should rather speed up execution when facing an unresponsive SUT. To evaluate the new suite settings, two test runs were conducted to compare old and new settings. The results show that fuzzing speed has improved slightly when many cases are skipped, but when most cases are passed the difference is negligible. Even the slight improvement was, however, not enough for running full runs for every product release candidate, hence the two separate pipelines.

Overall the work seems successful. Defensics was successfully taken as part of the existing CI and the brief comparison between automated and manual workflow shows that needed effort by a tester has clearly decreased. Although performance improvements were not enough for running full test case amount for every release candidate, full runs can still be executed on a weekly basis and triggered manually whenever needed. Other limitations mostly concern the level of automation. For example, target machines are expected to exist and be correctly configured before fuzz testing. The level of automation was therefore identified as the clear next step for the work. Other identified steps were related to suite configurations: more thorough analysis could be performed to further improve fuzzing performance and external instrumentation could be used for detecting more subtle errors.

REFERENCES

[1] Manes, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J. and Woo, M. The Art, Science, and Engineering of Fuzzing: A Survey. eng. (2018).

[2] Sorsa, S. *Protocol fuzz testing as a part of secure software development life cycle*. eng. Tietotekniikka – Pervasive Computing, 2018.

[3] Helin, A. *radamsa*. 2021. URL: `https://gitlab.com/akihe/radamsa` (visited on 02/09/2022).

[4] Oka, D. K., Makila, T. and Kuipers, R. Integrating Application Security Testing Tools into ALM Tools in the Automotive Industry. eng. *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2019, pp. 42–45. ISBN: 1728139252.

[5] Rossi, E.-P. Defensics-testaamisen automatisointi JYVSECTEC-ympäristöön. (2014).

[6] Patki, P., Gotkhindikar, A. and Mane, S. Intelligent Fuzz Testing Framework for Finding Hidden Vulnerabilities in Automotive Environment. eng. *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–4. ISBN: 9781538652572.

[7] Paananen, K., Seppa¨la¨, J., Koivisto, H. and Repo, S. Analysing security issues for a smart grid demonstration environment. eng. *22nd International Conference and Exhibition on Electricity Distribution (CIRED 2013)*. Vol. 2013. 615. Stevenage, UK: IET, 2013, pp. 1300–. ISBN: 1849197326.

[8] Suchorab, J., Staszkiewicz, K., Walkiewicz, J. and Dudek, M. Effective Fuzz Testing for Programmable Logic Controllers Vulnerability Research to Ensure Nuclear Safety. "International Conference on Nuclear Security 2020". 2020.

[9] Bratus, S., Hansen, A. and Shubina, A. LZfuzz: a fast compression-based fuzzer for poorly documented protocols. 2008.

[10] Woo, M., Cha, S. K., Gottlieb, S. and Brumley, D. Scheduling black-box mutational fuzzing. eng. *Proceedings of the 2013 ACM SIGSAC conference on computer & communications security*. CCS '13. ACM, 2013, pp. 511–522. ISBN: 9781450324779.

[11] Vieira, M., Antunes, N. and Madeira, H. Using web security scanners to detect vulnerabilities in web services. eng. *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 566–571. ISBN: 1424444225.

[12] Liang, H., Pei, X., Jia, X., Shen, W. and Zhang, J. Fuzzing: State of the Art. *IEEE transactions on reliability* 67.3 (2018), pp. 1199–1218. ISSN: 0018-9529.

[13] Li, J., Zhao, B. and Zhang, C. Fuzzing: a survey. eng. *Cybersecurity* 1.1 (2018), pp. 1–13. ISSN: 2523-3246.

[14] Wang, Y., Jia, P., Liu, L., Huang, C. and Liu, Z. A systematic review of fuzzing based on machine learning techniques. eng. *PloS one* 15.8 (2020), e0237749–e0237749. ISSN: 1932-6203.

[15] Munea, T. L., Munea, T. L., Lim, H., Lim, H., Shon, T. and Shon, T. Network protocol fuzz testing for information systems and applications: a survey and taxonomy. eng. *Multimedia tools and applications* 75.22 (2016), pp. 14745–14757. ISSN: 1380-7501.

[16] Standards, N. I. of and Technology. *NIST security vulnerability trends in 2020: an analysis*. Tech. rep. 2020. URL: https://www.redscan.com/news/nist-nvd-analysis/ (visited on 07/29/2021).

[17] Jahankhani, H., Watson, D. L., Me, G. and Leonhardt, F. *Handbook of electronic security and digital forensics*. eng. Singapore: World Scientific Publishing Co. Pte. Ltd, 2010. ISBN: 9789812837035.

[18] Ball, D., Simões Coelho, P. and Machás, A. The role of communication and trust in explaining customer loyalty. eng. *European journal of marketing* 38.9/10 (2004), pp. 1272–1293. ISSN: 0309-0566.

[19] Saaksvuori, A. *Product Lifecycle Management*. eng. 3rd ed. 2008. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 1-281-31438-2.

[20] Ruparelia, N. B. Software development lifecycle models. eng. *Software engineering notes* 35.3 (2010), pp. 8–13. ISSN: 0163-5948.

[21] Foundation, O. *OWASP Testing Guide v4*. 2014. URL: https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Testing_Guide_v4.pdf (visited on 06/07/2021).

[22] Felderer, M., Büchler, M., Johns, M., Brucker, A., Breu, R. and Pretschner, A. Security Testing: A Survey. Mar. 2016, pp. 1–51. ISBN: 9780128051580. DOI: 10.1016/bs.adcom.2015.11.003.

[23] Chess, B. *Secure programming with static analysis*. eng. 1st edition. Addison-Wesley software security series Secure programming with static analysis. Place of publication not identified: Addison Wesley, 2007. ISBN: 0-321-52035-1.

[24] Tassey, G. *The economic impacts of inadequate infrastructure for software testing*. 2002. URL: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf.

[25] Vicente Mohino, de, Higuera, B. and Montalvo, S. The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies. eng. *Electronics (Basel)* 8.11 (2019), pp. 1218–. ISSN: 2079-9292.

[26] Microsoft. *Microsoft Security Development Lifecycle (SDL)*. 2021. URL: https://www.microsoft.com/en-us/securityengineering/sdl/ (visited on 06/10/2021).

[27] Daud, M. I. Secure Software Development Model: A Guide for Secure Software Life Cycle. *Lecture Notes in Engineering and Computer Science* 2180 (July 2010).

[28] Standards, N. I. of and Technology. *National Vulnerability Database*. 2021. URL: https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all (visited on 06/01/2021).

[29] Bachmann, R. and Brucker, A. D. Developing secure software: A holistic approach to security testing. eng. *Datenschutz und Datensicherheit* 38.4 (2014), pp. 257–261. ISSN: 1614-0702.

[30] Dadeau, F., Fourneret, E. and Bouchelaghem, A. Temporal property patterns for model-based testing from UML/OCL. eng. *Software and systems modeling* 18.2 (2019), pp. 865–888. ISSN: 1619-1366.

[31] Al-Refai, M., Ghosh, S. and Cazzola, W. Model-Based Regression Test Selection for Validating Runtime Adaptation of Software Systems. eng. *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 288–298. ISBN: 1509018271.

[32] Yoo, S. and Harman, M. Regression testing minimization, selection and prioritization: a survey. eng. *Software testing, verification & reliability* 22.2 (2012), pp. 67–120. ISSN: 0960-0833.

[33] Mourad, A., Laverdière, M.-A. and Debbabi, M. Security hardening of open source software. eng. *Proceedings of the 2006 International Conference on privacy, security and trust*. PST '06. ACM, 2006, pp. 1–1. ISBN: 1595936041.

[34] Al-Mayahi, I. and Mansoor, S. P. ISO 27001 Gap Analysis - Case Study. eng. *Proceedings of the International Conference on Security and Management (SAM)* (2012), pp. 1–.

[35] Callahan, J. and Moretton, B. Reducing software product development time. eng. *International journal of project management* 19.1 (2001), pp. 59–70. ISSN: 0263-7863.

[36] Synopsys. *Fuzzing Guidelines*. URL: https://synopsys.skilljar.com/defensics-essentials/219275 (visited on 08/21/2021).

[37] Taipale, O., Kasurinen, J., Karhu, K. and Smolander, K. Trade-off between automated and manual software testing. eng. *International journal of system assurance engineering and management* 2.2 (2011), pp. 114–125. ISSN: 0975-6809.

[38] Duvall, P. M. *Continuous integration : improving software quality and reducing risk*. eng. Place of publication not identified, 2007.

[39] Foundation, R. F. *Robot Framework User Guide. Version 4.0.3*. 2021. (Visited on 07/01/2021).

[40] *ISO/IEC/IEEE 24765:2017. Systems and software engineering — Vocabulary*. Tech. rep. International Organization for Standardization, Sept. 2017. URL: https://www.iso.org/standard/71952.html.

[41] Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33. ISSN: 1545-5971.

[42] *ISO/IEC/IEEE 15288:2015. Systems and software engineering — System life cycle processes*. Tech. rep. International Organization for Standardization, May 2015. URL: https://www.iso.org/standard/63711.html.

[43] *ISO/IEC 27000:2018. Information technology — Security techniques — Information security management systems — Overview and vocabulary*. Tech. rep. International Organization for Standardization, Feb. 2018. URL: https://www.iso.org/standard/73906.html.

[44] Micskei, Z., Madeira, H., Avritzer, A., Majzik, I., Vieira, M. and Antunes, N. *Robustness Testing Techniques and Tools*. Ed. by K. Wolter, A. Avritzer, M. Vieira and A. van Moorsel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 323–339. ISBN: 978-3-642-29032-9. DOI: 10.1007/978-3-642-29032-9_16. URL: https://doi.org/10.1007/978-3-642-29032-9_16.

[45] Shahrokni, A. and Feldt, R. A systematic review of software robustness. *Information and Software Technology* 55.1 (2013). Special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010, pp. 1–17. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0950584912001048.

[46] *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. eng. Boston, MA, 2003.

[47] Feinbube, L. Software Fault Injection: A Practical Perspective. IntechOpen, 2018. ISBN: 9781789232585.

[48] Benso, A. and Di Carlo, S. The art of fault injection. *Control Engineering and Applied Informatics* 13.4 (2011), pp. 9–18. ISSN: 1454-8658.

[49] Koopman, P., Sung, J., Dingman, C., Siewiorek, D. and Marz, T. Comparing operating systems using robustness benchmarks. *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*. IEEE, 1997, pp. 72–79. ISBN: 9780818681776.

[50] Kropp, N., Koopman, P. and Siewiorek, D. Automated robustness testing of off-the-shelf software components. *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*. Vol. 1998-. IEEE, 1998, pp. 230–239. ISBN: 9780818684708.

[51] *Fuzzing for software security testing and quality assurance*. eng. Second edition. Artech House Information Security and Privacy Series. Boston, Massachusetts ; Artech House, 2018. ISBN: 1-63081-519-5.

[52] Chen, Y., Ian, T. and Venkataramani, G. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols. eng. *Proceedings of the 3rd ACM Workshop on forming an ecosystem around software transformation*. FEAST'19. ACM, 2019, pp. 17–23. ISBN: 1450368344.

[53] Miller, B., Fredriksen, L. and So, B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM* 33.12 (1990), pp. 32–44. ISSN: 0001-0782.

[54] Oehlert, P. Violating assumptions with fuzzing. *IEEE security & privacy* 3.2 (2005), pp. 58–62. ISSN: 1540-7993.

[55] Klees, G., Ruef, A., Cooper, B., Wei, S. and Hicks, M. Evaluating Fuzz Testing. *Proceedings of the 2018 ACM SIGSAC Conference on computer and communications security*. CCS '18. ACM, 2018, pp. 2123–2138. ISBN: 9781450356930.

[56] Miller, C., Peterson, Z. N. et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 4 (2007).

[57] Chen, C., Cui, B., Ma, J., Wu, R., Guo, J. and Liu, W. A systematic review of fuzzing techniques. eng. *Computers & security* 75 (2018), pp. 118–137. ISSN: 0167-4048.

[58] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S. The Oracle Problem in Software Testing: A Survey. eng. *IEEE transactions on software engineering* 41.5 (2015), pp. 507–525. ISSN: 0098-5589.

[59] Synopsys. *What is Fuzzing: The Poet, the Courier, and the Oracle*. 2017.

[60] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G. and Brumley, D. Optimizing seed selection for fuzzing. *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 861–875.

[61] Bohme, M., Pham, V.-T. and Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. eng. *IEEE transactions on software engineering* 45.5 (2019), pp. 489–506. ISSN: 0098-5589.

[62] Bounimova, E., Godefroid, P. and Molnar, D. Billions and billions of constraints: whitebox fuzz testing in production. eng. *Proceedings of the 2013 International Conference on software engineering*. ICSE '13. IEEE Press, 2013, pp. 122–131. ISBN: 1467330760.

[63] Initiative, J. T. F. T. *Assessing Security and Privacy Controls in Federal Information Systems and Organizations: Building Effective Assessment Plans*. Tech. rep. NIST Special Publication (SP) 800-53A, Rev. 4, Includes updates as of December 18, 2014. Gaithersburg, MD: National Institute of Standards and Technology, 2014. DOI: 10.6028/NIST.SP.800-53Ar4.

[64] Godefroid, P., Levin, M. Y. and Molnar, D. Automated Whitebox Fuzz Testing. Nov. 2008. URL: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[65] Ehmer, M. and Khan, F. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. eng. *International journal of advanced computer science & applications* 3.6 (2012). ISSN: 2158-107X.

[66] Doupé, A., Cova, M. and Vigna, G. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. eng. *Detection of Intrusions and Malware, and Vulnerability Assessment*. Vol. 6201. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–131. ISBN: 9783642142147.

[67] Bazzoli, E., Criscione, C., Maggi, F. and Zanero, S. XSS PEEKER: Dissecting the XSS Exploitation Techniques and Fuzzing Mechanisms of Blackbox Web Application Scanners. eng. *ICT Systems Security and Privacy Protection*. Vol. 471. IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2016, pp. 243–258. ISBN: 3319336290.

[68] Eddington, M. *Peach Fuzzer*. 2004. URL: https://peachtech.gitlab.io/peach-fuzzer-community/ (visited on 04/10/2021).

[69] Pham, V.-T., Boehme, M., Santosa, A. E., Caciulescu, A. R. and Roychoudhury, A. Smart Greybox Fuzzing. eng. *IEEE transactions on software engineering* (2019), pp. 1–1. ISSN: 0098-5589.

[70] Rash, M. and Wilk, J. *afl-cve. A collection of vulnerabilities discovered by the AFL fuzzer (afl-fuzz)*. 2017. URL: https://github.com/mrash/afl-cve (visited on 03/16/2021).

[71] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C. and Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. *NDSS*. Feb. 2017. URL: Paper=https://download.vusec.net/papers/vuzzer_ndss17.pdf%20Code=https://github.com/vusec/vuzzer64.

[72] Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D. and Rieck, K. Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. eng. *Security and Privacy in Communication Networks*. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Cham: Springer International Publishing, pp. 330–347. ISBN: 9783319288642.

[73] Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R. and Vigna, G. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. eng. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4176. New York: Springer, 2006, pp. 343–358. ISBN: 3540383417.

[74] Jones, D. *Trinity: Linux system call fuzzer*. 2010. URL: https://github.com/kernelslacker/trinity (visited on 05/09/2021).

[75] Wang, J., Chen, B., Wei, L. and Liu, Y. Skyfire: Data-Driven Seed Generation for Fuzzing. eng. *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594. ISBN: 9781509055333.

[76] Chen, T., Zhang, X.-s., Guo, S.-z., Li, H.-y. and Wu, Y. State of the art: Dynamic symbolic execution for automated test generation. eng. *Future generation computer systems* 29.7 (2013), pp. 1758–1773. ISSN: 0167-739X.

[77] Chipounov, V., Kuznetsov, V. and Candea, G. S2E: a platform for in-vivo multi-path analysis of software systems. eng. *SIGPLAN notices* 46.3 (2011), pp. 265–278. ISSN: 0362-1340.

[78] Newsome, J. and Song, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. Feb. 2005.

[79] Sparks, S., Embleton, S., Cunningham, R. and Zou, C. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. eng. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 477–486. ISBN: 0769530605.

[80] Tip, F. *A survey of program slicing techniques*. eng. 1994.

[81] Mitchell, T. M. *Machine learning*. eng. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997. ISBN: 0-07-042807-7.

[82] Chandramouli, S. *Machine Learning*. eng. 1st edition. Pearson Education India, 2018. ISBN: 93-89588-13-8.

[83] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F. and Whelan, R. LAVA: Large-Scale Automated Vulnerability Addition. eng. *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121. ISBN: 1509008241.

[84] Cha, S. K., Woo, M. and Brumley, D. Program-Adaptive Mutational Fuzzing. eng. *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741. ISBN: 1467369497.

[85] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. and Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. Jan. 2016. DOI: 10.14722/ndss.2016.23368.

[86] Jacobsen, O. and Lynch, D. *RFC 1208. A Glossary of Networking Terms*. Tech. rep. Mar. 1991. DOI: 10.17487/RFC1208. URL: https://www.rfc-editor.org/info/rfc1208.

[87] Tanenbaum, A. S. *Computer networks*. eng. 5th ed., international ed. Harlow: Pearson Education, 2014. ISBN: 1-292-02422-4.

[88] Alshmrany, K. and Cordeiro, L. Finding Security Vulnerabilities in Network Protocol Implementations. eng. (2020).

[89] IETF. *RFCs. Memos in the RFC document series contain technical and organizational notes about the Internet.* URL: https://www.ietf.org/standards/rfcs/ (visited on 04/22/2021).

[90] Postel, J. *RFC 793. Transmission Control Protocol*. Tech. rep. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793.

[91] Synopsys. *The Heartbleed Bug*. June 3, 2020. URL: https://heartbleed.com/ (visited on 04/23/2021).

[92] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M. and Halderman, J. The Matter of Heartbleed. eng. *Proceedings of the 2014 Conference on internet measurement conference*. IMC '14. ACM, 2014, pp. 475–488. ISBN: 1450332137.

[93] Tsankov, P., Dashti, M. and Basin, D. SecFuzz: fuzz-testing security protocols. eng. *Proceedings of the 7th International Workshop on automation of software test*. AST '12. IEEE Press, 2012, pp. 1–7. ISBN: 1467318221.

[94] Gorbunov, S. and Rosenbloom, A. AutoFuzz: Automated Network Protocol Fuzzing Framework. IJCSNS 10 (8 Aug. 2010). URL: http://paper.ijcsns.org/07_book/201008/20100836.pdf.

[95] Fang, K. and Yan, G. Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning. eng. *Computer Security*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 20–40. ISBN: 331998988X.

[96] Pham, V.-T., Bohme, M. and Roychoudhury, A. AFLNET: A Greybox Fuzzer for Network Protocols. eng. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465. ISBN: 9781728157788.

[97] Knudsen, J. and Varpiola, M. *Fuzz Testing Maturity Model*. Nov. 13, 2013.

[98] Synopsys. *Defensics User Guide. For Defensics 2020.12*. 2020.

[99] Synopsys. *Defensics Test Integration Guide. For Defensics 2020.12*. 2020.

[100] Case, J., Fedor, M., Schoffstall, M. and Davin, J. *RFC 1157. Simple Network Management Protocol (SNMP)*. Tech. rep. May 1990. DOI: 10.17487/RFC1157. URL: https://datatracker.ietf.org/doc/html/rfc1157.

[101] Synopsys. *Defensics Installation Guide. For Defensics 2020.12*. 2020.

[102] Synopsys. *Fuzzing Test Suites. Browse our library of 250+ pre-built fuzzing test suites by industry, technology, category, or keyword.* URL: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing/defensics.html (visited on 08/26/2021).

[103] *arp(7) — Linux manual page*. Aug. 13, 2020. URL: https://man7.org/linux/man-pages/man7/arp.7.html (visited on 09/11/2021).

[104] *icmp(7) — Linux manual page*. Nov. 26, 2017. URL: https://man7.org/linux/man-pages/man7/icmp.7.html (visited on 09/11/2021).