jamk.fi

Improving Crash Uniqueness Detection in Fuzzy Testing

Case JyvSectec

Mikko Pudas

Master's thesis
Month Year (March 2017)
Technology
Master's Programme in Cyber Security

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Pudas, Mikko	Master's thesis	March 2017
		Language of publication: English
	Number of pages: 89	Permission for web publication: x

Title of publication

Improving Crash Uniqueness Detection in Fuzzy Testing

Case JyvSectec

Degree programme

Master's Programme in Cyber Security

Supervisor(s)

Kotikoski, Sampo

Assigned by

JyvSectec, Silokunnas, Marko

Abstract

The purpose of fuzzing was defined to send malformed data in order to crash the program under test. Fuzzing has been defined as the most powerful test automation tool for discovering the security critical problems of software. JyvSectec ran into issues during one of the fuzzing sessions. When a program was fuzzed by AFL it indicated many unique crashes. A crash analysis was performed with the use Bash scripts, GDB, SQLite3 and a manual analysis pointed out that there were many almost identical crashes where the backtrace and even parameters were the same or almost same.

The task and objectives were set to replace the Bash script based crash analysis system by Python. One of the tasks before coding the Python program was to code a sample C program to be fuzzed by AFL. The Python program was implemented to carry out some clean-up of GDB outputs and Python program output results to a CSV database. After that, the crash analysis process in JyvSectec would have continued by SQLite3; howver, that part of the crash analysis was omitted.

Design research was used with intervention to measure the status before and after the use of the Python program to replace the Bash script based analysis method. Questionnaires were sent to all JyvSectec staff with structured and semi-structured questions.

The results of the intervention were mixed and indicated that satisfaction improved and the amount of manual work needed with the provided Python program decreased. However, separating different crashes remained an issue, and differentiating same or similar crashes and crash categorization were not improved by the intervention and the provided Python program.

Keywords/tags (subjects)

fuzzing, AFL, JyvSectec, crash analysis, American Fuzzy Lop, GDB

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Pudas, Mikko	Opinnäytetyö, ylempi AMK	Maaliskuu 2017
	Sivumäärä 89	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Improving Crash Uniqueness Detection in Fuzzy Testing

Case JyvSectec

Tutkinto-ohjelma

Master's Programme in Cyber Security

Työn ohjaaja(t)

Sampo Kotikoski

Toimeksiantaja(t)

JyvSectec, Marko Silokunnas

Tiivistelmä

Fuzzaus määriteltiin testattavan ohjelman kaatamiseksi viallista dataa käyttäen. Fuzzausta kuvailtiin tehokkaimpana testiautomaatiotyökaluna, jolla ohjelmiston turvallisuuskriittisiä ongelmia voitiin löytää. JyvSectec törmäsi yhdessä AFL-fuzzaustestaustapauksessa ongelmiin työkalun kaatumisten tunnistamisen suhteen. Fuzzattaessa AFL raportoi useita uniikkeja kaatumisia, mutta kun kaatumisia analysoitiin Bash scripteillä, GDBllä, SQLite3lla sekä manuaalisesti, oli uniikkeja kaatumisia vähän. Parametrit sekä niin sanottu GDBn backtrace olivat joko samoja tai lähes samoja, eli kaatumiset eivät olleetkaan uniikkeja toisin kuin AFL-työkalu tunnisti.

Tehtävänä sekä tavoitteena oli korvata Bash-skriptipohjainen kaatumisien analysointijärjetelmä Pythonilla koodatulla ohjelmistolla. Ennen Python ohjelmiston koodaamista C-kielinen sovellus koodattiin ongelman toistamiseksi ja sitä fuzzattin AFL-sovelluksella. Python-ohjelmisto koodattiin siivoamaan GDB-tulosteita duplikaattien tunnistamisen helpottamiseksi. Python-ohjelmistossa luotiin CSV-tietokanta, jota SQLite3-ohjelmisto käytti kaatumisien analysoinnissa. SQLite3-ohjelmistolla tehtävä kaatumisien analysointiosuus rajattiin opinnäytetyön ulkopuolelle.

Kehittämistutkimuksena interventiolla mitattiin tyytyväisyyttä ennen ja Python-ohjelmiston käytön jälkeen. Strukturoituja sekä puolistrukturoituja kysymyksiä sisältävät sähköpostikyselylomakkeet lähetettiin jokaiselle JyvSectecissä.

Tulokset olivat vaihtelevia, sillä tulosten mukaan tyytyväisyys kasvoi ja manuaalisen työn määrä väheni. Kuitenkin eri kaatumisien erottaminen oli yhä ongelma eikä samankaltaisten kaatumisien tai kaatumisien luokittelu parantunut interventiossa.

Avainsanat (asiasanat)

fuzzaus, AFL, JyvSectec, crash analysis, American Fuzzy Lop, GDB

Muut tiedot

Contents

1 Introduction....5
1.1 Thesis Background....5
1.2 Research Problem And Research Methods....7
1.3 Thesis Structure....9
2 Software Testing and Information Security....9
2.1 What Is Software Testing?....9
2.1.1 Software Quality Attributes....11
2.1.2 Software Testing Models....16
2.1.3 Software Testing Methods....18
2.1.4 The Role of Software Testing to the Information Security....23
2.2 Fuzzy Testing....25
2.2.1 Fuzzer Categories and Fuzzing Process....27
2.2.2 Types of Fuzzers and Sources of Data Used for Fuzzing....28
2.2.3 Fuzzy Testing with American Fuzzy Lop Fuzzer....31
2.3 Current Issues with Fuzzy Testing in JyvSectec....37
3 Thesis Implementation....37
3.1 Evaluation of Implementation Options....38
3.2 Information Gathering for the Chosen Approach....39
3.3 Crash Uniqueness Detection Challenges of the Chosen Approach....39
4 Crash Uniqueness Detection Intervention Results of the Chosen Approach and Conclusions....40
4.1 Crash Uniqueness Questionnaire Results before Intervention....40
4.2 Crash Uniqueness Questionnaire Results after Intervention....45
4.3 Crash Uniqueness Intervention Impact....50
5 Discussion....55

References 58

Appendices 60

Figures

Figure 1. The user interface of AFL. 32
Figure 2. Instrumenting software code for AFL Fuzzer. 33
Figure 3. Fuzzing with instrumented code. 33
Figure 4. AFL Fuzzing with failing test case as an input. 36
Figure 5. GDB crash dump from one of the inputs by AFL. 37

Tables

Table 1. An example of the content of store directory with a lot of timeouts. 35
Table 2. How satisfied is the respondent with the current process before intervention? 41
Table 3. How does the respondent see the amount of manual work before intervention? 42
Table 4. How easy it is to separate crashes before intervention? 43
Table 5. How much automated results processing should be used before intervention? 44
Table 6. The biggest issues in crash analysis before intervention. 45
Table 7. How satisfied is the respondent with the current process after intervention? 46
Table 8. How does the respondent see the amount of manual work after intervention? 47
Table 9. How easy it is to separate crashes after intervention? 48
Table 10. How much automated results processing should be used after intervention? 49
Table 11. The biggest issues in crash analysis after intervention. 50

Table 12. Satisfaction results before and after intervention. 51
Table 13. Amount of Manual work results before and after intervention. 52
Table 14. Results of how easy it is to separate different crashes before and after intervention. 53
Table 15. Results of how much automated results processing should be done before and after intervention. 54
Table 16. Results of theme categories about the biggest issues before and after intervention. 54

Acronyms and Abbreviations

AFL	American Fuzzy Lop
API	Application Programming Interface
Backtrace	A list of function calls indicating how program ended up where it is
BASH	Unix shell and command language created by Brian Fox
CSV	Comma separated values
GDB	The GNU Project Debugger
AFL-GCC	A tool that injects instrumentation for AFL when source code is available

1 Introduction

The purpose of this thesis was to evaluate and depict how crash dumps and uniqueness detection can be improved to require as little manual work as possible in JyvSectec. American Fuzzy Lop or AFL fuzzer is used as an example fuzzer and the program to be fuzzed is a simple self coded C program. Python is used to code a simple replacement tool for the current Bash script based process.

1.1 Thesis Background

Computer software is created by humans and humans make mistakes (Lagus 2013, 30). Before the Internet became mainstream, not much attention was paid to vulnerabilities since getting most of the limited resources was paramount (Lagus 2013, 30). According to Lagus (2013, 31), another issue with software vulnerabilities is the fact that security is often taken into consideration in the final development phases. Additionally, information systems are not simple and they are often connected to other complex information systems (Lagus 2013, 31).

According to Bhat (2015, 23), software testing is highly complex, yet, an imperative element of any software development life cycle. Software testing should be started as early as possible (Bhat 2015, 23); however, the costs of testing are high and Godefroid et al. (2008, 30) point out that usually testing accounts for about half of the R&D budget of many software development organizations.

Myers, Sandler & Badgett (2011, 5) state that in an ideal world every possible permutation of a program would be tested. However, in most cases that would not be possible or would need hundreds or thousands of possible input and output combinations and creating test cases for all the combinations would be impractical (Myers et al. 2011, 5). Positive testing is used to verify that the software works as it has been advertised whereas there is an effort to break the software in negative testing (Bhat 2015, 24).

Fuzzing has only one goal, to make the system crash (Takanen et al. 2008, 25). With fuzzing a large numbers of boundary cases are tested by either developers or quality assurance teams (Oehlert 2005, 58). The prime targets for fuzzing are input files, con-

figuration or registry entries, APIs, user interfaces, network interfaces, database entries or command line arguments (Oehlert 2005, 59).

Shapiro (2011, 58) points out that the fuzzing triggers race conditions, buffer overflows, failures to check return code and format or printf string problems. On the other hand, Takanen et al. (2008, 27) mention that fuzzing findings are at several levels where the field level finds overflows and integer anomalies. At the structural level, fuzzing finds underflows, repetition of elements and unexpected elements (Takanen et al. 2008, 27). At the sequence level, fuzzing finds out of sequence or omitted unexpected repetition or spamming of messages (Takanen et al. 2008, 27).

Fuzzing can also be a part of vulnerability analysis where fuzzing is used as a black box technique without the need of source code (Takanen et al. 2008, 102). Also whitebox fuzzing can be performed (Godefroid et al. 2012, 44). The trade-offs between whitebox and blackbox fuzzing are different, because blackbox fuzzing is simple, easy, lightweight and fast; however, may offer only limited code coverage (Godefroid et al. 2012, 44). Whitebox fuzzing is more complex but smarter (Godefroid et al. 2012, 44). For finding bugs, Godefroid et al. (2012, 44) point out that efficiency of either whitebox or black box fuzzing varies. A simple blackbox fuzzing is a good start if an application has never been fuzzed and after those bugs have been found it is time to use whitebox fuzzing (Godefroid et al. 2012, 44). The effectiveness of fuzzing bases on measuring how well fuzzing covers the input space of the tested interfaces and how good the used inputs are (Takanen et al. 2008, 27–28). Fuzzers that only generate random data based inputs are ineffective and find only naive programming errors (Takanen et al. 2008, 28). Godefroid et al. (2008, 32) point out that the ability of fuzzers to find bugs on low probability paths is limited. Above all fuzzing is about test automation to its fullest extent (Takanen et al. 2008, 136).

JyvSectec or Jyväskylä Security Technology at JAMK University of Applied Sciences was launched on September 2011 (JyvSectec). The purpose was to create one of the leading cyber security research, development and education centers in Finland and develop both national and international networking between various actors (JyvSectec). JyvSectec provides services related to software testing in order to detect functional weaknesses and deficiencies in their information security (JyvSectec). The

aim of the provided software testing is to avoid realization of damaging risks to the information systems (JyvSectec).

1.2 Research Problem And Research Methods

The objective of this master's thesis is to study how different crashes in fuzzy testing can be uniquely detected even if there are a several crash files. The current method requires a great deal of manual work, and many duplicate crash reasons are identified after the manual work. The objective is approached by studying what software testing and fuzzy testing are. The main questions to be answered are: Can crash uniqueness detection be improved compared to current Bash script based crash dump process that uses GDB within the JyvSectec case and can the current process that requires manual work be improved by the use of implemented Python program. Thus, by using automation the purpose of this design research is to demise the amount of manual work that JyvSectec has to use for analyzing the fuzzing results. As the author has been building a career as a software testing engineer, the purpose of this design research is to get familiar with the concept of fuzzing and thus increase the competence in that area of software testing.

The theoretical basis is based on design research where different options are evaluated in order to research how crashes can be uniquely detected in fuzzy testing and if crash uniqueness detection can be improved. The purpose of the design research is also to provide a simple Python program that can be used to automate the manually performed steps. Design research aims to make a change and Kananen (2015, 9) states that design research starts where the qualitative or quantitative research stops. On the other hand, Kananen (2015, 33) stresses that design research is not a research method on its own but instead design research consists of a group of research methods applied based on research case and methods. Thus, design research can be characterized as a blended methodology where qualitative and quantitative methodologies are blended (Kananen 2015, 33–34).

Because design research is quite often specific for certain organizations or companies, the issues are not unambiguous (Kananen 2015, 9). The research issue can be approached from various aspects, such as economic, legal or personnel points of

view, which leads to the diversity of aspects (Kananen 2015, 13). Different branches of science have different aspects, which again leads to different viewpoints of the research issue (Kananen 2015, 13).

Design research attempts to fix the problem that qualitative or quantitative research depicted (Kananen 2015, 9). Design research is characterized to be more than traditional quantitative or qualitative research that analyze the research issue, find out the causes and propose solutions (Kananen 2015, 40). Writing a report on what the research problem was, what the goals were, what was done and what the results were does not fulfill the definition of design research (Kananen 2015, 52). Instead, design research can be depicted as a phased process that consists of research and change phases that repeat each other (Kananen 2015, 42). The phases of research phase consist of understanding the current status, spotting the issues, defining the issues, searching for alternative solutions and evaluating solutions as well as choosing the correct solution (Kananen 2015, 42). The phases of change phase consist of implementation, evaluation and follow up phases (Kananen 2015, 42).

Providing a solution to the issue with design research will not ensure that the issue is no longer present (Kananen 2015, 13). The purpose of design research is to remove the issue instead of stating the reasons and ways how to tackle the issue (Kananen 2015, 13). Because design research is not a design methodology on its own, the evaluation of reliability and validity is challenging (Kananen 2015, 111). Thus, the evaluation of reliability and validity must be done based on the used methodology (Kananen 2015, 111).

Intervention data was gathered by semi-structured questionnaires before and after the intervention. The questionnaires were sent by email with approximately a two week responding time. The choice of using questionnaires was affected mostly by multiform teaching methods. The questions in the questionnaires were both open-ended, fixed and used Likert's attitude scale. Scales are used as scientific measuring instruments where questionnaires might simply gather information (Coolican 2009, 204). Research validity and reliability are evaluated in the discussion chapter.

Because design research is mostly performed for organizations and companies and the same phenomena can be approach from multiple aspects, the issues are not un-

ambiguous (Kananen 2015, 13). Similar design research about fuzzing and improving crash uniqueness detection was not found as a part of information research done in ZZRZV120 Asiantuntijan tiedonhankinta course. However, research about fuzzing has been conducted, and the essential references are Copeland (2003), Mili et al. (2015) and Takanen et al. (2008).

The Python program is meant to provide a solution to replace the current Bash script based process. GDB or GNU Project debugger is used by JyvSectec in debugging the crashes. The follow up consists of questionnaires that are done before a solution is tried and right after solution is tried.

1.3 Thesis Structure

The structure of design research follows the same structure as any other scientific paper (Kananen 2015, 15). Chapter two presents the theoretical background of software testing and the role of software testing with the information security. Chapter three presents current issues and evaluates different solution options, with an intervention implementation with Python. Chapter four includes intervention results and conclusions and chapter five an evaluation. Finally, the references and appendixes are mentioned.

2 Software Testing and Information Security

This chapter defines what is software testing, how it is performed, how software testing can attribute to the information security and what is fuzzy testing.

2.1 What Is Software Testing?

Copeland (2003, 2) points out that software testing has been defined in various ways and different people and organizations have various aspects concerning the purpose of software testing. Software testing as a part of software engineering is quite a young engineering discipline (Märijärvi et al. 1998, 11–12). The term software crisis was used by NASA in the 1960s to point out the unique properties of software engineering (Märijärvi et al. 1998, 12, 16). Mili et al. (2015, xiv) continue by pointing out

that the only engineering discipline, where product testing is a major technical and organizational concern, is software engineering. That is because of the size and complexity of software products, which also makes the design of software products more error prone (Mili et al. 2015, xiv). Another factor is the lack of standardized development processes for software products (Mili et al. 2015, xiv), which leads to using product controls instead of process controls in product quality control (Mili et al. 2015, xiv). Mili et al. (2015, xiv) mention the lack of practical and scalable static product analysis methods as a third factor, which leads to the use of dynamic methods. Other factors that affect software testing are changes during the software development or maintenance processes (Mili et al. 2015, xiv). Copeland (2003, 4) summarizes the many challenges that testing faces by listing those challenges. Those challenges are the lack of time to test properly or test well, too many input combinations to test, difficulties to determine the expected results of the tests or the lack of test oracles, rapidly changing or non existing requirements, no training in the processes related to testing, lack of tool support and management does not understand testing or does not care about quality (Copeland 2003, 4).

Pressman (2000, 426) emphasizes the importance of software testing by stating that it is a critical factor in software quality assurance activities. Haikala & Märijärvi (1998, 258) take another look at the software testing and define software testing as a planned approach to finding errors while the program under test or a part of it is run. Myers et al. (2011, 6) define testing as a "process of executing a program with the intent of finding errors." Kasurinen (2013, 10) states that software testing covers all the work performed in order to ensure that the software product will fulfill all the set requirements and that the completed work properties in the software product work as planned. Marcel et al. (2014, 632) point out that gaining confidence about a program's correctness by sampling its input space is the aim of automated program testing.

The terms verification and validation are often used to define software testing (Kasurinen 2013, 10). In verification, there are checks that ensure that the work-product will meet the requirements set out for it, and in validation, the focus is on evaluating the work product against user needs (Samaroo 2010, 38). Both verification and validation composes of several software quality assurance activities in addition

to performing software testing, such as reviews, quality and configuration audits, documentation reviews and installation testing (Pressman 2000, 467).

Pressman (2000, 466) states that a number of different testing strategies exist in the literature. Pressman (2000, 466) points out that all of those provide a template for testing and share some generic characteristics. Starting from the component level and working outwards towards integrating the whole computer-based system, Pressmann (2000, 466) stresses that different testing techniques are applicable at different points in time, testing is conducted by the developer of the software with an independent test group, and testing and debugging are different activities; however, debugging has to be accommodated in any testing strategy (Pressman 2000, 466). A testing strategy has to provide guidance for the practitioner and, for a manager, a set of milestones (Pressman 2000, 466). Also because of the deadline pressure, the progress has to be measurable and problems should surface as early as possible (Pressman 2000, 466). One of the testing models is the classic waterfall model where testing is just one of the phases in the software development process (Kasurinen 2014, 12). Testing checks that the developed software product has been implemented as designed and it fulfills the needs of the customer (Kasurinen 2014, 13). Testing will end when there are no major defects or the expected functionality is verified to work (Kasurinen 2014, 13). After testing is finished the software product is taken into use by the customer (Kasurinen 2014, 13). Some of the deficiencies of the waterfall model are the difficulty of gathering all the requirements and the role of testing as a quality control checkpoint after most of the software development has been completed (Kasurinen 2014, 14).

Mili et (2015, xiv) summarize software testing as an integral part of software quality assurance processes alongside other quality assurance techniques. Software testing complements static analysis techniques and software testing follows a formal goal-oriented stepwise process (Mili et al. 2015, xiv–xv).

2.1.1 Software Quality Attributes

Mili et al. (2015, 14) define several software quality categories that are closely linked to software testing. Also, inspections and reviews are critical to the development of high quality software (Takanen et al. 2008, 80). Haikala et al. (1998, 6) provide their

listing of software quality attributes. The size of the software and the amount of data processing can be illustrated by the lines of code or megabytes or the amount of functions or the sizes of the databases the software uses (Haikala et al. 1998, 6). Response time and real time requirements describe the software's ability to react (Haikala et al. 1998, 6). Response time measures the software's speed to react to inputs and real time requirements define the limits that the software must adhere to and, for instance, in some real time software requirements state that the response must not be too early or too late (Haikala et al. 1996, 6–7).

Reliability and decentralization are listed next, and Haikala et al. (1998, 7) stress the importance of reliability. Reliability is defined as protection from both software failures and external interference (Haikala et al. 1998, 7). Both defensive programming and redundant systems can be utilized (Haikala et al. 1998, 7). Haikala et al. (1998, 7) mention that more and more decentralization is also used.

Productization or tailorization level is the last quality attribute that Haikala et al. (1998, 7) mention. Tailored software is tailored for a certain specific purpose and the role of the customer is significant (Haikala et al. 1998, 7). Commercial off the shelf software product development develops software for the use of multiple customers (Haikala et al. 1998, 7). Haikala et al. (1998, 7) state that many software development projects align in the middle of those extremes (Haikala et al. 1998, 7).

Functional attributes depict how the software product under testing behaves based on inputs and outputs (Mili et al. 2015, 15). Functional attributes have a dependency to the specifications of the program and those depict what kind of situations the program is planned to face and what is the correct behavior for each of those situations (Mili et al. 2015, 15). Bath et al. (2014, 153) state that functional testing is the cornerstone of testing and if the software does not do what it is supposed to do, it makes no difference if the software is really fast or stunningly reliable. Mili et al. (2015, 15) divide functional attributes to those that are of a Boolean nature and of a statistical nature. Those attributes that are of Boolean nature are something that the software product either has or does not have (Mili et al. 2015, 15). Those attributes of statistical nature are something that the "software product has them to a smaller or larger extent" (Mili et al. 2015, 15).

Correctness and robustness are both attributes of a Boolean nature, and correctness refers to the behavior according to its specifications for all possible situations according to the domain defined in the specification (Mili et al. 2015, 15). Robustness refers both to the behavior according to its specification according to correctness and to the fact that the program behaves reasonably for situations outside the domain of the specification (Mili et al. 2015, 15).

Because correctness and robustness are difficult to define for any software products of realistic size, statistical attributes are used to measure how closely a software product is to being correct or robust (Mili et al. 2015, 15). Mili et al. (2015, 16) define dependability as a probability that for a period of operation time the system behaves according to its specifications. Dependability is further divided into reliability and safety (Mili et al. 2015, 16). Reliability refers to the probability that for a certain amount of time software product operates without violating its specifications (Mili et al. 2015, 16). Safety refers to the probability that for a certain amount of time the software product operates without causing a catastrophic failure (Mili et al. 2015, 16). Both reliability and safety are used to depict the software products' abilities to operate according to specifications whereras safety focuses on high-stakes clauses that might cause catastrophic losses such as losses of lives, mission criticalities or financial stakes (Mili et al. 2015, 16). Reliable software may fail seldom but may cause a catastrophic loss and a software system may be safe but totally unreliable (Mili et al. 2015, 16).

In addition to dependability, security is another attribute of statistical nature (Mili et al. 2015, 16). Security refers to voluntary actions committed by malicious sources whereas dependability refers to system design flaws that may cause failures (Mili et al. 2015, 16). Mili et al. (2015, 16) divide security into four aspects such as, confidentiality, integrity, authentication and availability. Confidentiality depicts a system's ability to prevent unauthorized access to confidential data (Mili et al. 2015, 16). Integrity depicts a system's ability to prevent losses or damages to critical data (Mili et al. 2015, 16). Authentication depicts a system's ability to identify users and grant permissions (Mili et al. 2015, 16). Availability depicts a system's ability to continue serving its user communities and it is measured as a percentage (Mili et al. 2015, 16).

Operational attributes depict the operational conditions of the software (Mili et al. 2015, 15). Latency, throughput, efficiency, capacity and scalability are examples of operational attributes (Mili et al. 2015, 16). Latency depicts the time between the submissions of queries and the responses and varies based on system's workload (Mili et al. 2015, 16). Throughput depicts the volume of processing the system delivers per unit of operational time (Mili et al. 2015, 16). Efficiency depicts the software's ability to deliver its functions with as few computational resources as possible (Mili et al. 2015, 16). Capacity depicts the number of simultaneous users the system can serve with a certain amount of quality of service (Mili et al. 2015, 16). Scalability is used to depict the ability of the system to continue delivering service even when the workload is exceeding its original capacity (Mili et al. 2015, 16–17).

Usability attributes depict how the software product can be used and adapted to the needs of the user (Mili et al. 2015, 15). Mili et al. (2015, 18) identify five attributes that are ease of use, ease of learning, customizability, calibrability and interoperability. Ease of use refers to qualities that support the ease of use such as simplicity of system interactions, uniformity of interactions, availability of help menus and tolerance of misuse (Mili et al. 2015, 18). Ease of learning refers to qualities that support ease of learning by intuitive system interactions, consistency of interaction protocols and uniformity of system outputs (Mili et al. 2015, 18). Customizability refers to software's functional abilities to be tuned based on particular end user's requirements and the more control the end user has, the better is the customizability of the software (Mili et al. 2015, 18). Calibrability refers to the abilities to tune the software to the specific operational requirements the end user has (Mili et al. 2015, 18). Interoperability refers to the abilities to work in conjunction with other applications in collaboration (Mili et al. 2015, 18).

Business attributes depict the development costs, the costs of using and evolving the software product (Mili et al. 2015, 15). Development costs, maintainability, portability and reusability are business attributes (Mili et al. 2015, 19). Development costs are an important attribute and, for instance, person-months invested into the development of the software product can be calculated (Mili et al. 2015, 19). Maintainability refers to the amount of effort that is invested in the maintenance of the product after the delivery of the product (Mili et al. 2015, 19). Portability refers to the aver-

age costs of porting the product from hardware or software platforms to other platforms (Mili et al. 2015, 19). Reusability refers to potential benefits and means the ability of the software product to be reused either as a part or in whole (Mili et al. 2015, 19).

Structural attributes depict the internal structure of the software product (Mili et al. 201, 15). Structural attributes are of interest of technical personnel such as engineers and designers (Mili et al. 2015, 20). Mili et al. (2015, 20) list four structural attributes, design integrity, modularity, testability and adaptability. Qualities of good design integrity include simplicity, orthogonality, economy of concept, cohesivess of the design, consistency of design rules and adherence to simple design disciplines (Mili et al. 2015, 20).

Modularity refers to information hiding and the main principles of modular design are separation between the module specification and its implementation (Mili et al. 2015, 20). Cohesion and coupling are two attributes related to modularity (Mili et al. 2015, 20). Bath et al. (2014, 421) point out that strongly coupled modules usually require more development and testing effort. Strong cohesion, on the other hand, is a desirable design attribute because then a module implements one specific part of functionality (Bath et al. 2014, 421).

Testability refers to the extent of testing the system or components to an arbitrary level of thoroughness (Mili et al. 2015, 20). Controllability and observability are two attributes of testability and controllability is the bandwidth of input values that can be submitted as test data to the component by controlling system inputs (Mili et al. 2015, 20). Observability refers to the extent that the component's output can be inferred by observing the system output (Mili et al. 2015, 20).

Adaptability refers to the ease how software can be modified to changing requirements (Mili et al. 2015, 21). Mili et al. (2015, 21) point out that adaptability differs from customizability by referring to changes to the system requirements. Another difference is that whereas customizability refers to changes that are carried out by the end user, adaptability refers to changes made by software engineers (Mili et al. 2015, 21).

2.1.2 Software Testing Models

Software testing is a part of phases in software development life cycle model (Haikala et al. 1998, 23). Software development life cycle is a structure with the aim of development of a software product (Khan, M. E & Khan, F. 2014). Same methods and tools are used in all software development including agile and spiral software development models (Takanen et al. 2008, 78). Also Takanen et al. (2008, 78) point out that software development rarely follows straightforward processes.

According to Haikala et al. (1998, 25), the most common software life cycle model at that time was waterfall model, where software development is split into phases. After the development of each component has been finished, the software components have been integrated into a system, and the software testing phase is started (Kasurinen 2013, 13). The software testing phase ends, and the software product is ready to be used when there are no significant errors present or, at least, the software product fulfills the requirements set by the customer (Kasurinen 2013, 13). Cripsin & Gregory (2009, 15) describe the waterfall model as gated and the gatekeepers may block the progress of project if set milestones are not met. The major weaknesses of waterfall model are the unrealistic approach because rarely exact requirements are known, many adjustments are quite often needed, and the waterfall model with few checkpoints leads to unpredictability and makes managing the software development difficult (Kasurinen 2013, 22–23).

V model is developed to put more emphasis on software testing (Kasurinen 2013, 14). Black (2007, 24) defines V model as a refinement of waterfall model that puts more focus on testing. Testing is no longer a separate phase in software development life cycle, instead, testing is performed for each phase (Kasurinen 2013, 14). Unit testing is performed against software code, integration testing is performed against technical specifications, system testing is performed against functional specification and acceptance testing is performed against requirements specifications (Samaroo 2010, 38). The weakness of V model is the fact that it is quite often both schedule and budget-driven (Black 2007, 24). When either time or money starts to run out, mostly testing is cut back because most of the testing occurs at the end (Black 2007, 24). Kasurinen (2013, 14–15) points out that even in V model testing is

started too late, for instance, the requirements are created way before testing is started.

Evolutionary and incremental models have been developed to cope with the challenges of the V model (Black 2007, 25). Clearly defined increments are used where a system is analyzed, designed, developed and tested (Black 2007, 25). The project team can deliver at least some portion of the planned functionality at any point after the first increment is tested (Black 2007, 25). The incremental model is followed when the set of features are defined up front where as the evolutionary model is followed when over time the set of features is evolving (Black 2007, 25). The level of formality varies from lightweight Extreme Programming models to Rapid Application Development models (Black 2007, 25).

The spiral model is most useful when there is no way of specifying the system that is to be built, however, it must deliver the right set of features at the end (Black 2007, 28). Haikala et al. (1998, 32) uses term prototyping and points out that prototyping is quite useful when the user interface is defined. The weakness of using spiral model or prototyping is the fact that the software may look like it is finished even though the most of the work is still not done (Haikala et al. 1998, 33).

Lastly, Black (2007, 28) points out a sarcastic code and fix model where the development of the system is performed without the real idea of what is being built. A final product is set to be built without using the prototype, and the process may consist of writing and debugging the code without any unit testing (Black 2007, 28). Code is sent to people without professional skills for testing where many bugs are found (Black 2007, 29). These bugs are fixed live on the test environment by programmers without checking the code in version controlling systems (Black 2007, 29). The same bug is found again and the bug is again fixed on the test environment and process is repeated again and again with little coordination until money, time or patience of the project team are exhausted and the system is either released or cancelled (Black 2007, 29).

2.1.3 Software Testing Methods

Software testing can be based on static analysis or dynamic analysis (Takanen et al. 2008, 79). Static analysis does not involve the execution of the code and can be performed in addition to source code also on procedures or on architectural designs (Bath et al. 2014, 265). One of the major benefits of static analysis is the possibility to perform testing early (Bath et al. 2014, 266). Static analysis also makes code more maintainable and more portable and is essentially a cost-effective activity (Bath et al. 2014, 266). Kasurinen (2013, 65) points out that any defects found during static analysis are a lot cheaper to fix than failures found in dynamic testing. Static analysis also supports reviews of the same items because the results from static analysis may indicate parts to focus attention in reviews (Bath et al. 2014, 266). The limitations of static analysis are its ability to find actual defects because static analysis lists areas of code that need further investigation to check whether a defect is present (Bath et al. 2014, 267). Also, a tool support is needed because static analysis may become complex if software is anything more complex than Hello World (Bath et al. 2014, 267).

Dynamic analysis is the opposite of static analysis (Kasurinen 2013, 65) and it is performed while the software is executing (Takanen et al. 2008, 79). Takanen et al. 2008, 79) point out that combining both of these approaches is a sign of good test process.

The methods of testing can also be partitioned differently (Takanen et al. 2008, 79). Takanen et al. (2008, 79) quote Sommerville's thoughts of dividing testing into validation and defect testing. Validation testing shows that the software functions according to user requirements (Takanen et al. 2008, 79–80). Defect testing uncovers flaws in the software instead of simulating its operational use and tries to find inconsistencies between the system and its specifications (Takanen et al. 2008, 80).

Another division of testing is based on access to the source codes and black box testing and white box testing terms are used (Takanen et al. 2008, 80). Black box testing bases testing on the requirements and specifications and requires no knowledge of the internal structure of the code under test (Copeland 2003, 8). White box testing on the other hand is based on the internal structures, paths and implementation of the software under test and requires detailed programming skills (Copeland 2003, 8).

Kasurinen (2013, 65) points out that black box testing is the most traditional form of testing. The inputs are provided and software's or system's outputs are checked without checking what occurs inside the software's logic (Kasurinen 2015, 65). Copeland (2003, 20) states that no knowledge of system under test's internal paths, structures or implementation is needed. The role of testing is to check that outputs are what they were supposed to be (Kasurinen 2014, 66). Copeland (2003, 23) describes several black box testing techniques such as equivalence class testing, boundary value testing, decision table testing, pairwise testing, state-transition testing, domain analysis testing and use case testing. Bath et al. (2014, 68) add cause-effect graphing, user story testing and combinatorial testing as specification based techniques.

Equivalence class testing is used to reduce the number of test cases while reasonable level of test coverage is maintained (Copeland 2003, 24). Test conditions are grouped into partitions that will be treated the same way (Bath et al. 2014, 68). Boundary value testing focuses on the boundaries where many defects hide (Copeland 2003, 40). Data structures, such as tables, memory and disk capacity are usually of a fixed size and boundary value testing tests what would happen on the border and when the border is increased (Bath et al. 2014, 78). Decision tables are used to document multifaceted business rules that the system implements and decision tables support creation of test cases (Copeland 2003, 59).

Pairwise testing may be used when the number of test combinations is very large (Copeland 2003, 90). Instead of testing all combinations for all the values for all the variables, pairwise testing tests all pairs of variables, thus significantly reducing the number of needed test cases (Copeland 2003, 90).

State-transition testing uses state-transition diagram to capture some types of system requirements and to document the internal design of the system (Copeland 2003, 94). State-transition diagrams document both the incoming and processed events and the responses by the system (Copeland 2003, 94). State-transition diagrams help to identify states, events, actions and transitions that should be tested (Copeland 2003, 110).

Domain analysis testing is a testing technique where efficient and effective test cases are identified when several variables are or should be tested together (Copeland 2003, 116). Domain analysis testing builds on and generalizes equivalence class and boundary value testing and is useful because multiple variables are tested simultaneously (Copeland 2003, 123).

Use case testing uses use cases to define scenarios that depict how an actor uses the system to accomplish certain goals (Copeland 2003, 128). Scenario is a sequence of steps that depict the interactions the actor and the system has (Copeland 2003, 128). The actor is generally humans but other system can also be actors (Copeland 2003, 128).

White box testing differs from black box testing by paying attention to the inputs and outputs while paying attention to the inner logic of the system under test (Kasurinen 2014, 67). The inputs are selected to execute selected paths (Copeland 2003, 140). Copeland (2003, 140) stresses that white box testing is more than code testing, it is path testing that can be applied to test the paths between modules within subsystems. Kasurinen (2014, 68) mentions that several metrics can be used, such as code coverage or used paths. Disadvantages of white box testing are that it is unable to detect bad requirements or missing properties (Kasurinen 2014, 68). Also number of execution paths can get so large that it cannot be tested (Copeland 2003, 141). Data sensitivity errors may not be detected, for instance, p=q/r; may not execute correctly when r equals 0 (Copeland 2003, 141). Finally, the tester must possess programming skills to understand and evaluate the system under test and Copeland (2003, 141) mentions that many testers lack programming skills.

Copeland (2003, 144) mentions control flow and data flow testing techniques as white box testing techniques. Control flow testing identifies the executions paths of a module and executes test cases to cover those paths (Copeland 2003, 165). Copeland (2003, 147) states several levels of coverage of control flow testing. Bath et al. (2014, 298–299) use statement testing, decision testing, condition testing, decision condition testing and multiple condition testing instead of coverage levels.

Structured testing or basis path testing identifies test cases based on the analysis of the topology of the control flow graph (Copeland 2003, 154). Control flow graph con-

sists of nodes and edges that have been derived from software module (Copeland 2003, 154). Cyclomatic complexity is calculated and cyclomatic complexity is the minimum number of independent, not looping paths that generate all possible program paths of that module (Copeland 2003, 155). Pressman (2000, 436) define cyclomatic complexity as a quantitative measure of the logical complexity of a program. Calculated cyclomatic complexity in basis path testing defines the number of independent paths and presents the upper limit of the number of tests that has to conducted in order ensure that all statements have been executed at least once (Pressman 2000, 436). Cyclomatic complexity of a graph is the number of edges minus the number of nodes added two (Copeland 2003, 154). If all decisions of the control flow graph are binary and there are p amount of binary decision, then cyclomatic complexity is the value of p added one (Copeland 2003, 155). For the Appendix 1 program, the CCCC (Littlefair) counts the cyclomatic complexity of 9 as shown in Appendix 2. After the cyclomatic complexity has been calculated, a set of basis paths is done and test case for each basis path is created and those test cases are executed (Copeland 2003, 154).

In statement testing every executable statement is executed at least once (Bath et al. 2014, 299). Decision testing is more intrusive and it requires that all true and false decision outcomes are tested (Bath et al. 2014, 300). Condition testing tests that every atomic condition results in true or false outcomes (Bath et al. 2014, 302). But condition testing does not ensure that decision outcome is also reached, but decision condition testing requires that both false and true decision conditions are tested so that both true and false decision outcomes are reached (Bath et al. 2014, 304). Multiple condition testing consider all possible combinations of true and false decision outcomes for all the individual atomic conditions within a decision point and theoretically 2^n test cases are needed to cover "^{n}" atomic conditions (Bath et al. 2014, 304).

Data flow testing technique shows the processing flow thorough the module and also the definitions, uses and destructions of each of the variables (Copeland 2003, 171). Copeland (2003, 177) states that while testing a module or system under test, paths through the module should be enumerated. Then, for each variable at least one test case should be created to cover every define-use pairs (Copeland 2003, 177). Call graphs can be used to see the big picture of the program's architecture (Bath et al.

2014, 278). Call graphs also show the internal module structure (Bath et al. 2014, 276).

Gray box testing is an approach where a peek is taken into the internal structure of the system under test to understand how it has been implemented and that knowledge is used to choose more effective black box tests (Copeland 2003, 8). Kasurinen (2013, 68) states that gray box testing combines the best of both white and black box testing. Khan et al. (2012, 14) mention as advantages of gray box testing the benefits of combining both white box and black box testing techniques, instead of relying on the source code, the tester relies on interface definitions and functional specifications and can design excellent test scenarios. Lastly Khan et al. (2012, 14) point out that unbiased testing is done from users' point of view. The disadvantages of gray box testing are the limited test coverage and many program paths will remain untested (Khan et al. 2012, 14).

Bath et al. (2014, 293) point out that by combining first black box test techniques and the testing is then supplemented by white box techniques, it provides the optimal level of coverage effectiveness in the fastest time. By using only black box techniques relatively quickly a certain level of coverage can be reached whereas using only white box techniques it takes more investment at first and after a while higher level of coverage is reached (Bath et al. 2014, 293).

Third way of dividing testing is based on the test types (Samaroo 2010, 49). Samaroo (2010, 49) presents four categories starting from functional testing, non-functional testing, structural testing to testing after changes have been made to the code. Functional testing focuses to determine whether the software does what it is supposed to do (Bath et al. 2014, 153). Also the scope of functional testing changes based on the level of the development cycle and when doing unit testing, the focus is on the functionalities of the individual unit but when integration testing is being done, the focus is on various interfaces (Bath et al. 2014, 153).

Non-functional testing mostly uses black box testing techniques and tests the behavioral point of view of the system (Samaroo 2010, 50). Structural testing focuses on the structural aspects of the system, such as the code or architectural definitions of the system (Samaroo 2010, 50). Testing related to code after changes have been

made consists of retesting and regression testing (Samaroo 2010, 50). The focus of retesting is to confirm that the existed problem has been removed and the focus of regression testing is to ensure that no additional defects were introduced because of the changes that were made (Samaroo 2010, 50).

Arcuri et al. (2012, 274) state that guidelines are needed about where and when a particular testing should be used because it is unlikely that a certain testing method is going to be the best option for all testing problems.

2.1.4 The Role of Software Testing to the Information Security

Software testing related to security and reliability was in its infancy state until the late 1990s (Takanen et al. 2008, 22). Yet, Takanen et al. (2008, 71) point out that software quality issues, bugs such as, programming flaws or design flaws, are the main reasons behind many software vulnerabilities. Codefroid, Levin & Molnar (2012, 40) point out that many security vulnerabilities occur in code for file and packet parsing. Stephens et al. (2016, 1) state that even with new techniques for memory corruption and execution redirection mitigations, those flaws still account for over a third of all vulnerabilities found.

New automated vulnerability analysis systems have been designed that can be categorized to static, dynamic and concolic analysis systems (Stephens et al. 2016, 1). Static analysis systems can show that a certain piece of code is secure but those are imprecise causing a lot of false positives and cannot provide actionable input that triggers the detected vulnerability (Stephens et al. 2016, 1). Fuzzers and other dynamic analysis systems identify application flaws by monitoring the execution and can provide needed inputs to trigger the flaws (Stephens et al. 2016, 1). The disadvantage of dynamic analysis systems is the need for input test cases that may require a lot of manual effort (Stephens et al. 2016, 1) Concolic execution engines utilize interpretation of the program and generate inputs with the aim of constraint solving techniques in order to explore the state spaces of the binary for triggering the vulnerabilities (Stephens et al. 2016, 1). The disadvantage of concolic execution engines is path explosion that limits the scalability of concolic execution engines (Stephens et al. 2016, 1–2).

Testing software does not present an easy task, and standard software testing techniques focus on the correctness of software (DeMott 2006, 7). Fuzzing focuses on finding exploitable software bugs (DeMott 2006, 7). Takanen et al. (2008, 102) point out that fuzzing is not trying to verify or validate a system but rather tries to find defects with the goal to uncover as many vulnerabilities as possible. DeMott (2006, 7) stresses the importance of both types of testing by stating that there's a problem if the software doesn't do what it is supposed to do and there is a different kind of problem if the software has exploitable vulnerabilities. Chess et al. (2007, 9) point out that in practice, most of the software quality efforts focus on testing program functionality where the purpose is to find the bugs that affect most users in the worst ways. However, for the purpose of finding security problems comparing the implementation to the requirements is inadequate, and it is almost impossible to improve software security by just improving quality assurance (Chess et al. 2007, 9).

The advantage of fuzzing is that it can take place earlier than traditional vulnerability assurance practices, which take place in the late phases of the software development life cycle (Takanen et al. 2008, 71–72). Traditional vulnerability assurance practices focus on protecting from known attacks and identify known vulnerabilities from existing system (Takanen et al. 2008, 72). Fuzzing, symbolic execution and taint analysis are three major vulnerability detection techniques (Cai et al. 2014, 231). The purpose of fuzzing is to find new undetected flaws (Takanen et al. 2008, 72). Symbolic execution uses symbolic values as program inputs and illustrates the values of variables as symbolic expressions of those inputs. While analyzing the program, symbolic execution can theoretically find all possible execution paths (Cai et al. 2014, 232). Software testing can also get good code coverage by using symbolic execution (Cai et al. 2014, 232). The disadvantages of symbolic execution are path explosion, where there may be an extreme number of paths, path-divergence where computing the precise path constrains is challenging and complex-constraint where constraint solver fails to find solutions to complex yet satiable path constraints (Cai et al. 2014, 232). The focus of taint analysis is on variables that can be modified by users and those variables can become security vulnerabilities when used to execute potentially dangerous commands (Cai et al. 2014, 232). Taint analysis detects most of the input validation vul-

nerabilities but suffers from slow execution and false negatives with execution paths (Cai et al. 2014, 232).

Fuzzers are not silver bullets, which find all security problems (Takanen et al. 2008, 135). At its best, fuzzing is a proactive technique for catching vulnerabilities before others find and exploit those (Takanen et al. 2008, 135). Also DeMott et al. (2007, 2) point out that formal engineering practices, solid quality assurance or full code audit and penetration testing is not replaced by fuzzing.

2.2 Fuzzy Testing

Takanen et al. (2008, 1) point out that the purpose of fuzzing is to send malformed data to the system under test in order to crash it to reveal reliability issues. Fuzzing is defined as a highly automated testing technique that covers several boundary cases by use of invalid data as inputs to ensure the absence of exploitable vulnerabilities in the system under test (Takanen et al. 2008, 1). The term fuzzing originates from modem applications' tendency to fail because of random inputs that were caused by line noise on fuzzy telephone lines (Takanen et al. 2008, 1). Cai, Yang, Men & He (2014, 231) categorize fuzzing as a traditional vulnerability detection technique that was first proposed by Barton Miller in 1989.

Both black box and white box fuzzing can be performed (Godefroid, P., Kiezun, A. & Levin, M). Black box fuzzing sends randomly modified well-formed inputs but can utilize grammars to generate well formed inputs and add application-specific knowledge to guide the generation of input variants (Godefroid et al.). White box fuzzing executes the program under test with well formed inputs both concretely and symbolically (Godefroid et al.) Constraints on program inputs are created during the execution of conditional statements (Godefroid et al.). Those constraints show how inputs are used by the program and different control paths are executed based on inputs that are defined by satisfying assignments for the negation of each constraint (Godefroid et al.). This process is repeated for the newly created inputs in order to execute all feasible control paths of the system under test (Godefroid et al.). Also grey box fuzzing exists (DeMott, J., Enbody, R. & Punch, WF. 2007, 1). DeMott et al. (2007, 3) state that it is possible to monitor the running executable with as much

detail that a debugger will permit without access to source code directly. The advantage of grey box fuzzing is its effectiveness to find bugs that capture replay mutation black box tools do not (DeMott et al. 2007, 1).

According to Takanen et al. (2008, 2) fuzzing is the most powerful test automation tool for discovering software's security critical problems because when fuzzing, a software crash is a crash and therefore, there are no false positives (Takanen et al. 2008, 2). Also fuzzing works against any applications that process inputs regardless of used programming languages (Takanen et al. 2008, 63). Instead of establishing completeness or correctness fuzzing complements traditional testing by combining the power of randomness, protocol knowledge and attack heuristics to discovering untested combinations of code and data (DeMott 2006, 1).

Vulnerabilities are easily found be software testers, developers and researches by triggering malformed or malicious inputs via standard interfaces (Takanen et al. 2008, 7). Fuzzing is generally performed as black box testing (Takanen et al. 2008, 2) and because it is essentially functional testing, it can be performed in several steps during the software development and testing (Takanen et al. 2008, 17).

Fuzzing is one form of robustness testing and it complements both feature and performance testing (Takanen et al. 2008, 18). Robustness testing is defined as an ability to tolerate exceptional inputs and stressful environmental conditions and it tries to fulfill the negative testing requirements by using either random or semi-random inputs (Takanen et al. 2008, 18–19). With fuzzing the security of any process, service, device, system or network, can be tested regardless what specific interfaces are supported and no matter what exact interfaces are supported (Takanen et al. 2008, 26). Fuzzing is more interested in how system under test behaves compared to which components are used to build it (Takanen et al. 2008, 21).

Chess et al. (2007, 11) point out feeding fuzzing feeds randomly generated inputs to the program under test and testing with purely random inputs tends to inefficiently repeat the same conditions again and again. Without proper iteration refinement the fuzzer spends most of time fuzzing only a small part of the program's states (Chess et al. 2007, 11).

2.2.1 Fuzzer Categories and Fuzzing Process

Takanen et al. (2008, 26) categories fuzzers based on several criteria. One of those categories includes fuzzers based on the application area where they are used and what attack vectors they support (Takanen et al. 2008, 26). Different injection vectors are used but some fuzzers provide a general-purpose framework (Takanen et al. 2008, 26). Some fuzzers can be used to test both servers and clients (Takanen et al. 2008, 26).

Another categorization is based on test case complexity (Takanen et al. 2008, 26). Various layers of the target software can be targeted and different test cases penetrate different layers in the application's logic (Takanen et al. 2008, 26). Static and random template based fuzzers test only simple request-response protocols or file format fuzzers without dynamic functionality (Takanen et al. 2008, 27). Takanen et al. (2008, 29) state that if fuzzer supplies totally random characters, those are in general very inefficient and will not find many bugs. Dynamic generation or evolution based fuzzers may not understand the protocol or file format that is fuzzed, but will learn based on the feedback from the target system (Takanen et al. 2008, 27). Model-based or simulation-based fuzzers implement the tested interface and in addition to message structures also unexpected messages in sequences are generated (Takanen et al. 2008, 27). Model-based fuzzers can emulate protocols or file format interfaces almost completely and this allows them to penetrate deeper and exercise the parsing and input handling routines thoroughly reaching even the state machines and output generation routines and thus uncover more vulnerabilities (Takanen et al. 2008, 28–29).

Fuzzers typically have protocol modeler, anomaly library, attack simulation engine, runtime analysis engine, reporting and documentation functionalities (Takanen et al. 2008, 29–30). Protocol modeler contains functionality related to data formats and message sequences (Takanen et al. 2008, 29). The simplest protocol modeler's use message templates (Takanen et al. 2008, 29). Takanen et al. (2008, 29) state that some fuzzers might include a sample of known inputs to trigger vulnerabilities but some fuzzers use random data. Attack simulation engine utilizes a library of attacks or anomalies to generate the actual fuzz test cases (Takanen et al. 2008, 29).

Runtime analysis engine monitors the performance of the system under test and reporting functionalities may provide detailed reports or some may not provide reports at all (Takanen et al. 2008, 29). Documentation functionalities make the use of fuzzing tools easier (Takanen et al. 2008, 30).

A simplified view of fuzz testing is sending sequences of messages to the system under tests (Takanen et al. 2008, 30). After those changes in the system under test and received messages are analyzed but fuzzing is more than just sending and receiving messages (Takanen et al. 2008, 30). First tests are generated, then sent to the system under tests and the target is continuously monitored in order to catch and record failures (Takanen et al. 2008, 30–31). Lastly, the pass fail criteria are defined with the goal to perceive errors as they happen (Takanen et al. 2008, 31).

2.2.2 Types of Fuzzers and Sources of Data Used for Fuzzing

Takanen et al. (2008, 145) state that there are endless ways and tools to perform fuzzing. Most of the time the used test cases to fuzz utilize a library of known heuristics or mutate a sample input (Takanen et al. 2008, 137). A generation based fuzzer generates its own semi-valid sessions where mutation fuzzer takes a known good sample and mutates the sample to create semi-valid sessions (Takanen et al. 2008, 137). Usually mutation fuzzers are either generic fuzzers or general purpose fuzzers (Takanen et al. 2008, 137). But mutation fuzzers understand nothing about the underlying format (Takanen et al. 2008, 138). Terms intelligent or dumb fuzzers are used to indicate the level of interface knowledge a fuzzer has (Takanen et al. 2008, 144). Fully dumb fuzzer randomly flips bits in a file and that fuzzing results to lower code coverage (Takanen et al. 2008, 144). Intelligent fuzzer may understand what each field in the file represents and changes those according to specifications and no invalid data or options will be used (Takanen et al. 2008, 144). Takanen et al. (2008, 144) point out that most fuzzers are somewhere in-between.

Both generation based fuzzers or general purpose fuzzers may use randomized approach or deterministic approach and some tools attempt to include both approaches (Takanen et al. 2008, 139). In randomized approach one fuzzer randomly picks either valid or invalid commands, adds them randomly to a test case and chooses random data for the arguments of those commands (Takanen et al. 2008, 138). An-

other fuzzer might use more deterministic approach and fuzzer might have a sample of invalid data in the library with a series of command sequences (Takanen et al. 2008, 138–139). That invalid data is supplied in some repeatable manner with each command (Takanen et al. 2008, 139). Takanen et al. (2008, 139) stress that generally the more deterministic approach to fuzzing will outperform the randomized approach. Stephens et al. (2016, 2) point out that multiple analysis techniques can be combined in order to leverage their strengths while mitigating their weaknesses. A fuzzer could be used to explore initial compartments of the application and concolic execution engine could be used to guide the fuzzer to the next compartment (Stephens et al. 2016, 2).

In addition to generation based fuzzers or mutation fuzzers, single-use fuzzers are fuzzers quickly created for a certain task (Takanen et al. 2008, 145). Single-use fuzzers can be used if one size fits all generic fuzzers cannot be tuned to fulfill the requirements of a specific application (Takanen et al. 2008, 146). Fuzzing frameworks have a set of routines that can be used to write a fuzzer with the fundamental idea of code reuse (Takanen et al. 2008, 146). For instance, Fuzzled has helper functions that allow fuzzing tools to be developed (Takanen et al. 2008, 148). Protocol specific fuzzers are developed for a certain protocol (Takanen et al. 2008, 148). For instance, ftp-fuzz is designed to fuzz FTP servers (Takanen et al. 2008, 149).

Generic fuzzers can be used to test several interfaces or applications (Takanen et al. 2008, 149). File fuzzer flipping bits in any file types is an example of generic fuzzer (Takanen et al. 2008, 149). Capture-replay fuzzers operate by obtaining a known good communication file or typical argument and then modify it and repeatedly deliver it to the system under test (Takanen et al. 2008, 150). The goal of capture-replay fuzzers is to quickly fuzz either new or unknown protocol and capturing provides partial interface definition (Takanen et al. 2008, 150). In in-memory fuzzing the arguments are modified in memory before they are used in the programs' functions (Takanen et al. 2008, 161). In-memory fuzzing is more suited to closed source applications (Takanen et al. 2008, 161). Because the targeted functions may not be available from users input, a reverse engineer is needed to identify the start and stop locations of parsing routines that are to be fuzzed (Takanen et al. 2008, 161). The advantage of in-memory fuzzing is that the chosen functions can be fuzzed without un-

derstanding how they function even when those functions would not be reached by generic fuzzers because they would not be available from users' input (Takanen et al. 2008, 162).

Takanen et al. (2008, 162) provide another classification of fuzzers based on the interfaces they test. Local program fuzzers fuzz command line arguments, environment variables and other interfaces exposed (Takanen et al. 2008, 162). File fuzzing is also another example of local program fuzzing (Takanen et al. 2008, 162). Network interfaces fuzzer sends semi-valid application packets either to server or client (Takanen et al. 2008, 162). File fuzzing sends semi-valid audio files, video files or any file types to the application under test for parsing (Takanen et al. 2008, 163). API fuzzing focuses supplying unexpected parameters to the called function either with or without access to the source code (Takanen et al. 2008, 164). Web fuzzing is often used to refer to fuzzing of form fields of web applications and finding all valid pages, URL pages and inputs (Takanen et al. 2008, 164). Client-side fuzzers focus on fuzzing the client side instead of server side (Takanen et al. 2008, 164). Takanen et al. (2008, 164) point out that in the past client-side testing has not been done much. Finally, Takanen et al. (2008, 165) present OSI layer 2 through 7 fuzzing where any of the layers are fuzzed.

Takanen et al. (2008) state that SCADA and industrial platforms have received little testing from security community. Shapiro, Bratus, Rogers & Smith (2011, 57) state that applying current fuzz-testing techniques are difficult to apply to SCADA systems because of the use of proprietary protocols. Another factor that makes fuzzing of SCADA systems difficult are time-sensitiveness and session oriented nature of many SCADA systems (Shapiro et al. 2011, 58). Also attaching fuzzer and debugger to the target systems may not be possible with SCADA systems (Shapiro et al. 2011, 58).

Data used for fuzzing can be created from test cases by cycling through a protocol, randomly inserting data or from a library of known attacks (Takanen et al. 2008, 140–141). A fuzzer that uses test cases has some amount of test cases that are run against supported target protocol (Takanen et al. 2008, 140). The same tests are sent every time the fuzzer is run and a fuzzer of this kind is closely related to automated test tools or stress testing tools (Takanen et al. 2008, 140). Typically, a fuzzer that uses test cases is a generation fuzzer (Takanen et al. 2008, 140).

Fuzzer that cycles through a protocol by inserting some type of data to send semi-valid input in another way to generate used data for fuzzing (Katanen et al. 2008, 140). Cycling through results to a deterministic number of runs (Katanen et al. 2008, 140). Another way to create data for fuzzing is when fuzzer randomly keeps inserting data for a certain time period (Takanen et al. 2008, 140). If seeded by the user, these random fuzzers can be repeatable (Takanen et al. 2008, 141).

If a library of known and useful attacks are used, each variable to be tested is fuzzed with each type of attack by using random, weighted or deterministic order, priority and pairing of this search (Takanen et al. 2008, 141).

2.2.3 Fuzzy Testing with American Fuzzy Lop Fuzzer

American Fuzzy Lop Fuzzer or AFL Fuzzer is a security focused brute-force fuzzer that can be used either in compile time instrumentation mode or in traditional blind fuzzer mode (Zalewski 2016). Instrumentation can be either done either while compiling or by the use of QEMU hypervisor (Stephens et al. 2016, 5).

American Fuzzy Lop uses a modified form of edge coverage in order to pick up small, local-scale changes to program control flow (Zalewski 2016). Input generation is done by a genetic algorithm, mutating inputs based on the genetics inspired rules and ranking them by a fitness function (Stephens et al. 2016, 5). Fitness functions base on unique code coverage where an execution path is triggered, which is different from the paths triggered by other inputs (Stephens et al. 2016, 5). Union of control flow transitions, which American Fuzzy Lop has seen from its inputs, such as tuples of the source and destination basic blocks are tracked by American Fuzzy Lop (Stephens et al. 2016, 5). The inputs that make an application execute in a different way get prioritized in the generation of future inputs (Stephens et al. 2016, 5). In order to reduce the size of the path spaces for loops, American Fuzzy Lop uses a heuristic approach where only log(N) paths are considered for each loop instead of N paths (Stephens et al. 2016, 5). Randomization of the programs interferes with the genetic fuzzer's evaluation of inputs because an input, which produces interesting paths under a certain random seed may not do so under another random seed (Stephens et al. 2016, 5). If randomization is not removed, the fuzzing component is likely to explore only few paths, but if constant randomness is used, then the program accepts the same input

each time and that allows the fuzzer to find this value and subsequently explore further (Stephens et al. 2016, 5).

The process of American Fuzzy Lop is to load user-supplied initial test cases into the queue, then take next input file from the queue, attempt to trim the test case to the smallest size, which will not change the measured behavior of the system under test, mutate the file repeatedly by using a variety of traditional fuzzing strategies and if any of the generated mutations caused new state transitions that were recorded by the instrumentation, new entry of the mutated output is added to the queue and then the algorithm takes next input file from the queue and repeats (Zalewski 2016).

Blind fuzzer mode is used with -n parameter pointing to the program to be fuzzed after core dumps are instructed to be outputted as files. For the Lubuntu, the command sudo bash -c 'echo core.%e.%p > /proc/sys/kernel/core_pattern' has to be typed every time a system is restarted. After that the user interface of American Fuzzy Lop is shown and it can be seen in Figure 1 that non-instrumented mode is in use.

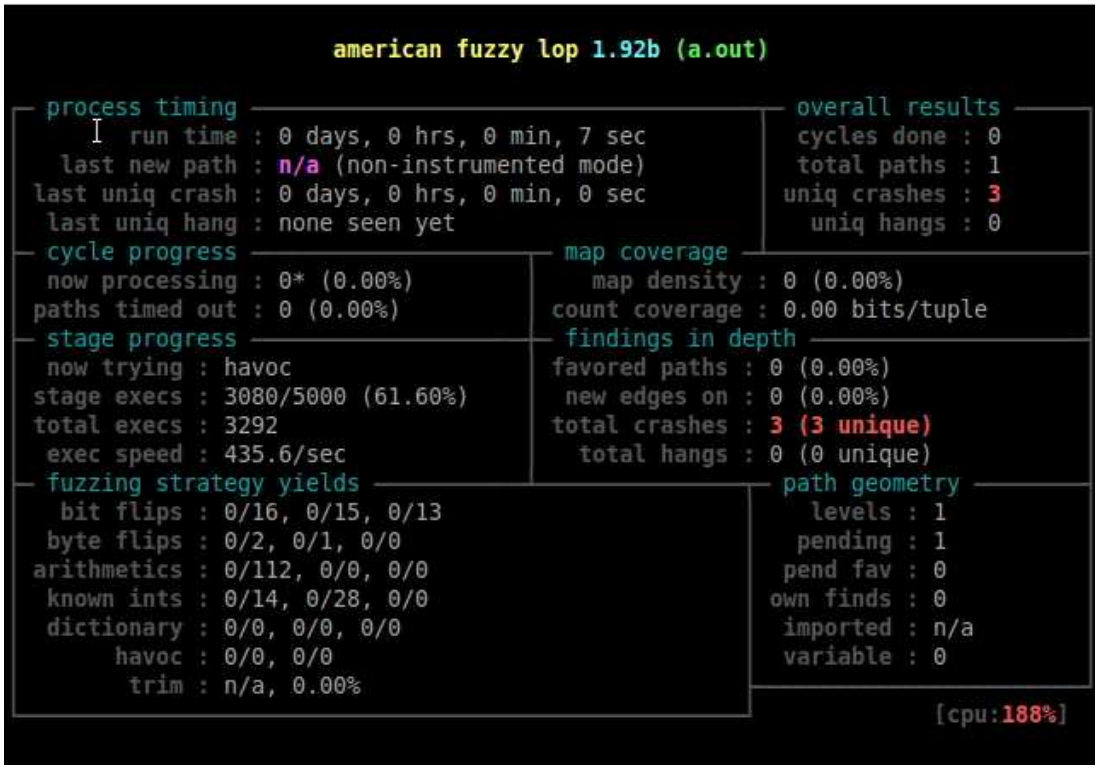


Figure 1. The user interface of AFL.

If instrumentation is to be used, the fuzzed program has to be instrumented with afl-gcc (Zalewski 2016). The instrumentation will print how many locations were instrumented, see in Figure 2.

Figure 2. Instrumenting software code for AFL Fuzzer.

During instrumentation assembly code is injected to the target program that is used to trace executions paths as new inputs are entered (Margaritelli 2015). Injected assembly code is also used to determine if known or unknown execution paths is triggered by a new mutation input (Margaritelli 2015). When fuzzing with instrumented code, last new path should show the time when last new path was found like in Figure 3 (Zalewski 2016).

Figure 3. Fuzzing with instrumented code.

The difference between instrumented and non-instrumented afl-fuzzing can be demonstrated using the same afl-test.c program attached to Appendix 1. When fuzzing is done without instrumentation total paths is 1 and total crashes is 0. When fuzz-

ing is done with instrumentation total paths is 5 and total crashes was 63 with 1 unique.

American Fuzzy Lop fuzzes until Ctrl-C is pressed but at least one queue cycle should be completed before fuzzing is stopped (Zalewski 2016). Completing one queue cycle may take from seconds to even a week (Zalewski 2016). The fuzzing is performed by afl-fuzz utility that requires a read-only directory with initial test cases, a directory to store results and path to the binary to be fuzzed (Zalewski 2016). For example, when command ./afl-fuzz -i in2 -o out2 /home/virtual/Työpöytä/afl-1.92b/a.out is used, the -i parameter points out to a directory with initial test cases and -o parameter points out to a directory to store the fuzzing results (Zalewski 2016). American Fuzzy Lop comes with sample test cases containing small standalone files that can be used to seed afl-fuzz (Zalewski 2016). The archives directory has, among others, samples of rar, tar and zip (Zalewski 2016). Images directory has, among others, samples of bmp, jpeg and png (Zalewski 2016). Multimedia directory has a sample of h264 and others directory has among others samples of js, pdf, rtf and text files (Zalewski 2016).

The directory to store results will have three subdirectories that are updated in real time (Zalewski 2016). Queue directory has test cases for every distinctive execution paths and the starting files given by the user (Zalewski 2016). Crashes directory has unique test cases that caused the program to receive a fatal signal and the entries are grouped by the received signal (Zalewski 2016). Hangs directory has unique test cases that cause the tested program to time out (Zalewski 2016). American Fuzzy Lop considers crashes and hangs unique if the associated execution paths involve any state transitions that have not been seen in previously recorded faults (Zalewski 2016). Crash is considered unique if the crash trace includes not seen a tuple in any of the previous crashes or if the crash trace is missing a tuple that was present every time in earlier faults (Zalewski 2016).

An example of both how provided input is modified and the contents of hangs of store directory is in Table 1 when a text based initial test case speed.txt with content of 56 in ASCII or 0011 0101 0011 0110 in binary format was used. At that time the non-instrumented fuzzed program used while loop until a valid input of driver's speed was provided with timeout set to 25 seconds in afl-fuzz. That causes several

timeouts to occur instead of crashes and the contents of timeout directory was over 500 files.

Table 1. An example of the content of store directory with a lot of timeouts.

File name in the hangs directory	File contents in Binary	File contents in Hex
id:000000,src:000000,op:flip1,pos:0	1011 0101 0011 0110	0xb536
id:000001,src:000000,op:flip1,pos:0	0111 0101 0011 0110	0x7536
id:000002,src:000000,op:flip1,pos:0	0001 0101 0011 0110	0x1536
id:000003,src:000000,op:flip1,pos:0	0010 0101 0011 0110	0x2536
id:000004,src:000000,op:flip1,pos:0	0011 1101 0011 0110	0x3d36
id:000005,src:000000,op:flip1,pos:0	0011 0001 0011 0110	0x3136
id:000006,src:000000,op:flip1,pos:1	0011 0101 1011 0110	0x35b6
id:000007,src:000000,op:flip1,pos:1	0011 0101 0111 0110	0x3576
id:000008,src:000000,op:flip1,pos:1	0011 0101 0001 0110	0x3516
id:000009,src:000000,op:flip1,pos:1	0011 0101 0010 0110	0x3526
id:000010,src:000000,op:flip1,pos:1	0011 0101 0010 1110	0x352e

In order to ease crash analysis American Fuzzy Lop fuzzer has a crash exploration mode where a crashed test case is provided as an input and American Fuzzy Lop uses its uses its genetic algorithms to see how far can be reached within the instrumented codebase while the program is kept in the crashing state (Zalewski 2016). Figure 4

shows an example where the crashes are used as inputs and the Appendix 1 program keeps crashing, however, only one unique crash is identified.

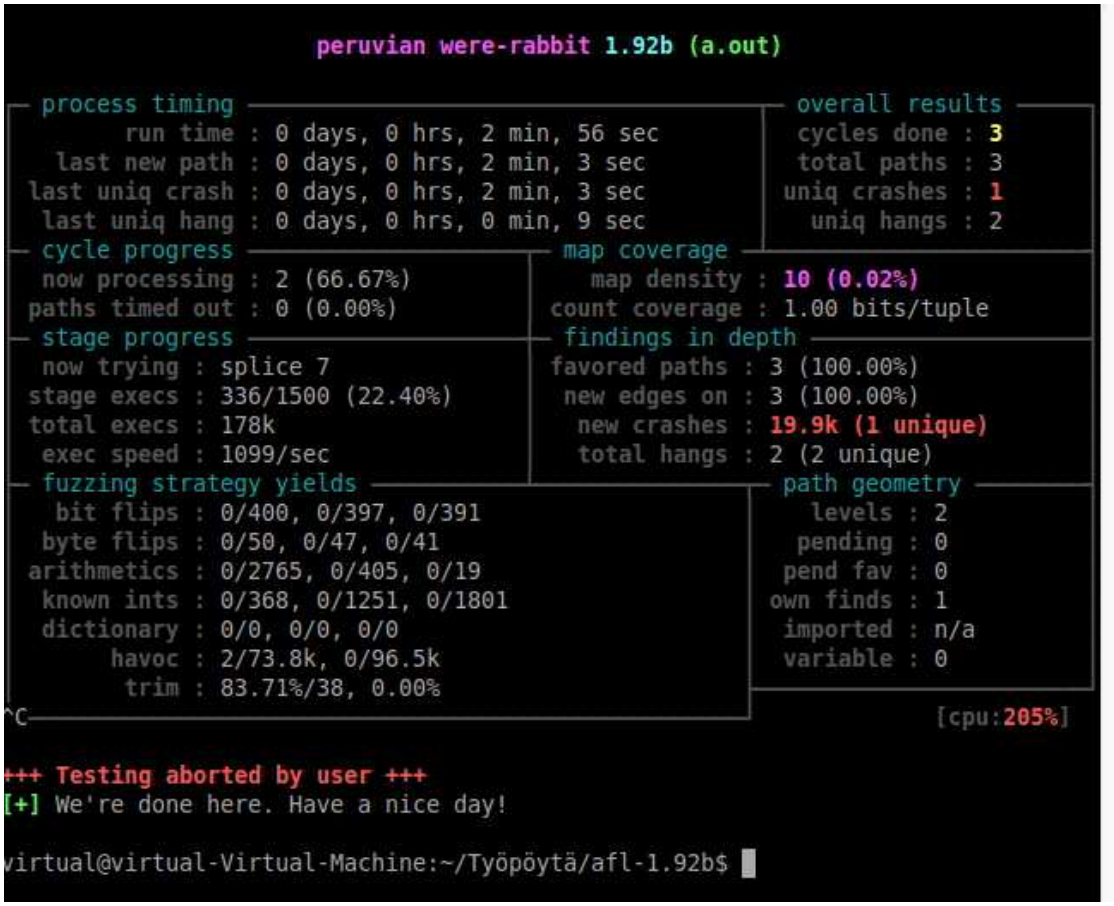


Figure 4. AFL Fuzzing with failing test case as an input.

American Fuzzy Lop produces a coverage-based grouping of crashes that can be triaged manually or use GDB scripts to analyze (Zalewski 2016). Also, every crash can be traced to its parent non-crashing test case in the queue, which should make it easier to detect faults (Zalewski 2016). Zalewski (2016) points out that some crashes caused by fuzzing can be quite difficult to evaluate for exploitability without lots of work in debugging and code analysis.

If Appendix 1 program is run under GDB or The GNU Project Debugger and the unique crash output of crash exploration is input, a buffer overflow will occur when breakpoint has been set to line 25 as seen in Figure 5.

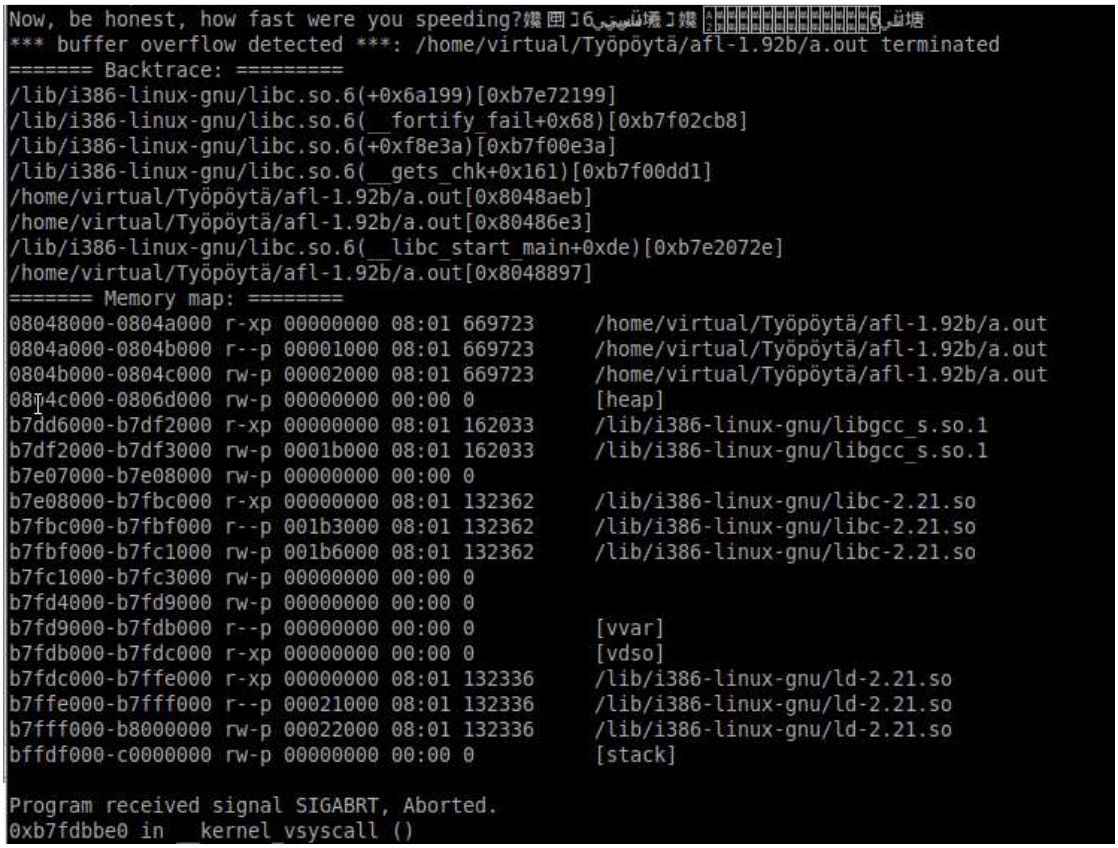


Figure 5. GDB crash dump from one of the inputs by AFL.

2.3 Current Issues with Fuzzy Testing in JyvSectec

JyvSectec or Jyväskylä Security Technology at JAMK University of Applied Sciences was launched in September 2011 (JyvSectec). JyvSectec provides services related to software testing in order to detect functional weaknesses and deficiencies in their information security (JyvSectec). There has been one occasion where the results of American Fuzzy Lop indicated thousands of unique crashes; however, when results were manually analyzed by the use of GDB the crashes were almost the same with the same backtraces and occasionally with the same parameters. In order to diminish the amount of manual work, a script based method is in used that saves backtraces, crashing file and crash signals to a SQLite3 database where searches are made.

3 Thesis Implementation

The implementation part of this thesis focuses on answering the research questions. The fuzzed program is coded with C language and program to replace the manual analysis work is coded using Python.

3.1 Evaluation of Implementation Options

JyvSectec's current process related to analysis of fuzzing requires a great deal of manual work. The aim of this design research is set to implement a program to ease that process, and the research objective of this master's thesis is approached by implementing the fuzzed program with enough nodes that the AFL fuzzer's instrumentation can be used instead of dump black box fuzzing. The C program is a simple speeding program that has several branches depending on the answers. The C program uses gets function that Chess et al. (2007, 155) depict as "the most widely acknowledged buffer overflow pitfalls". Chess et al. (2007, 155) state that the problems with gets function are so acknowledged that some compilers warn if gets is used and GCC gives a warning when Appendix 1 C program is compiled.

Because fuzzing the speeding C program without instrumentation only found one path and in order to provide more realistic approach, instrumentation is used. Zalewski (2016) states that instrumentation can be injected by a tool that functions as a replacement for GCC. Zalewski (2016) continues by stating that the GNU compiler collection includes front ends for C, C++, Objective-C, Fortran, Ada and Go. Out of those languages the author is familiar with C so that is the choice of the programming language for the fuzzed example program. Another reason for the choice of C language is mentioned by Chess et al. (2007, 14) where a generic defect is defined as a "problem that can occur in almost any program written in the given language". A buffer overflow is a security problem and also an excellent example of generic problems in C language (Chess et al. 2007, 14).

The automation part of the crash analysis is coded with Python as agreed with JyvSectec. The Python program is a simple program and outputs both Excel's xls and CSV files that are used as a database. Comma separated values can be imported to SQLite3 for further processing. The other options that were evaluated were real database implementations such as ODBC based implementation. Those options were rejected, and especially the ODBC based implementation would have taken a great amount of resources compared to the current implementation without giving any benefits. Python program uses xlsxwriter library provided by McNamara (2016) and xlrd library provided by Python Software Foundation. The functionality of the Python

program is verified to reproduce the same issue that JyvSectec run into. Crash analysis is done and even though AFL detects the crashes in Appendix 6 as unique manual crash analysis points out that the primary reason might be the same.

Lubuntu 14.04 LTS is the choice of both testing and implementation environment because AFL is not available for Windows. Lubuntu environment is used both for coding the C program and Python program.

3.2 Information Gathering for the Chosen Approach

As design research is conducted, the researcher can act also as an observer (Kananen 2015, 52). In order to answer the main research questions if crash uniqueness detection can be improved compared to the current status and if the current manual process can be automated, an intervention is performed. The intervention is performed by sending questionnaires (in Appendix 4) to measure current process satisfaction, and the questionnaires (Appendix 5) are sent with Python program (in Appendix 3) to measure satisfaction with proposed solution. Thus, the difference between previous state T_1 and state after intervention T_2 is measured (Kananen 2015, 55).

Both questionnaires contain five multiple choice questions, and the last question is semi-structured question. Likert's attitude scale is used for four questions and both questionnaires are attached as Appendices. Because the amount of persons involved in intervention is only two, and both work for JyvSectec and possess fuzzing related experience, background questions are not included in the questionnaires.

T_1 questionnaire was sent during mid January 2017 and T_2 questionnaire was sent during early February 2017 with approximately two weeks reserved for answering. The questionnaire was sent to all people in JyvSectec and n consists of two persons. The response rate for T_1 questionnaire is 100 percent and for T_2 is 100 percent.

3.3 Crash Uniqueness Detection Challenges of the Chosen Approach

The Python program's outputs are used to evaluate crash uniqueness detection by SQLite3. Appendices 6 shows an example output after the C program (in Appendices 1) is fuzzed as AFL detected three unique crashes. If a quick search is done, those

crashes share many common features, such as SIGABRT, raise.c:55 or gets_chk.c:67. However, #8 is different in every crash and in crash#1 afl_test3.c line 38 is where the crash occurred. However, in every crash file gets_chk.c line 67 is present pointing out that reading input may have been the cause of each crash. The main challenge is how to automate a duplicated crash analysis when the crash dumps and their content varies in different situations. However, the process of defining when a crash is unique or not is done by JyvSectec and it is out of the scope of this design research.

In order to verify that coded Python program can be used to automate crash analysis instead of Bash script Libjpeg 6b is also fuzzed as non-instrumented and instrumented. Libjpeg fuzzing results in eleven unique crashes by AFL but manual crash analysis might indicate that only two separate crashes exist.

4 Crash Uniqueness Detection Intervention Results of the Chosen Approach and Conclusions

Intervention results before and after intervention with the Python program for crash analysis are presented in this chapter with conclusions. Kananen (2015, 30) points out that by combining research results and conclusions unnecessary repetition can be avoided.

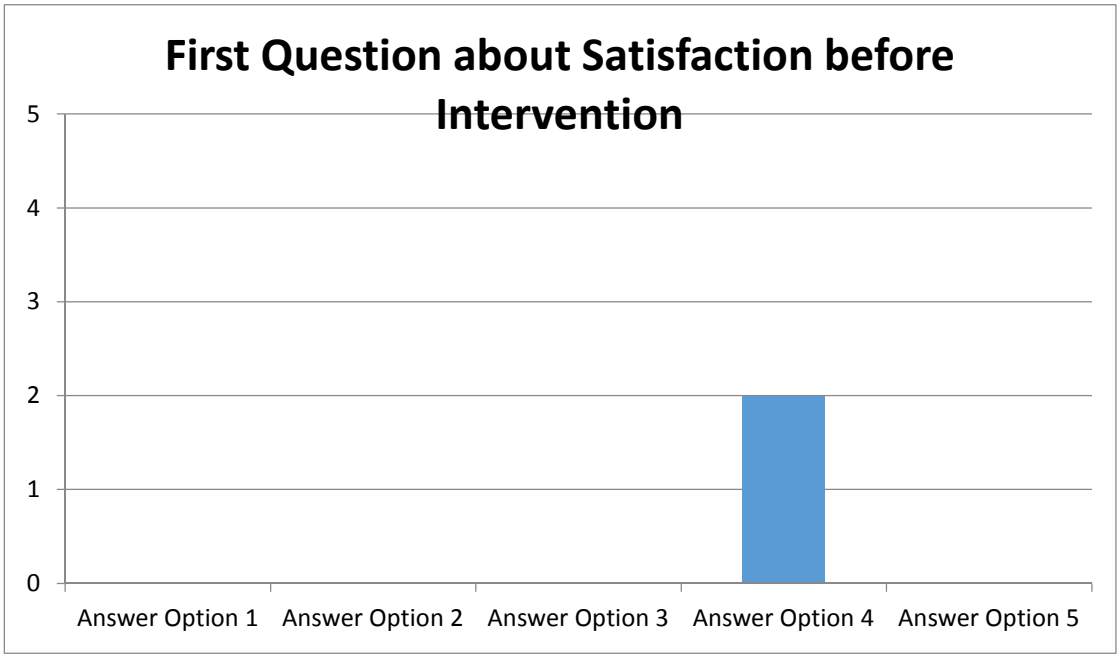
4.1 Crash Uniqueness Questionnaire Results before Intervention

Answers to the research questions about crash uniqueness detection improvement in JyvSectec and if the current process requiring manual work can be improved by the use of implemented Python program are provided. Quantitative research method is used to present the results of the questionnaires. The results of the questionnaires are converted into tables that represent the number of answers from all the returned answers (Kananen 2015, 101). Likert's scale is used to illustrate the answer options except for the fifth answer where an analysis and synthesis are made from the answer. The questionnaires are added as a part of Appendices.

The first question focuses on satisfaction with the current process of analyzing AFL crash results before intervention. The first question asks how satisfied the respond-

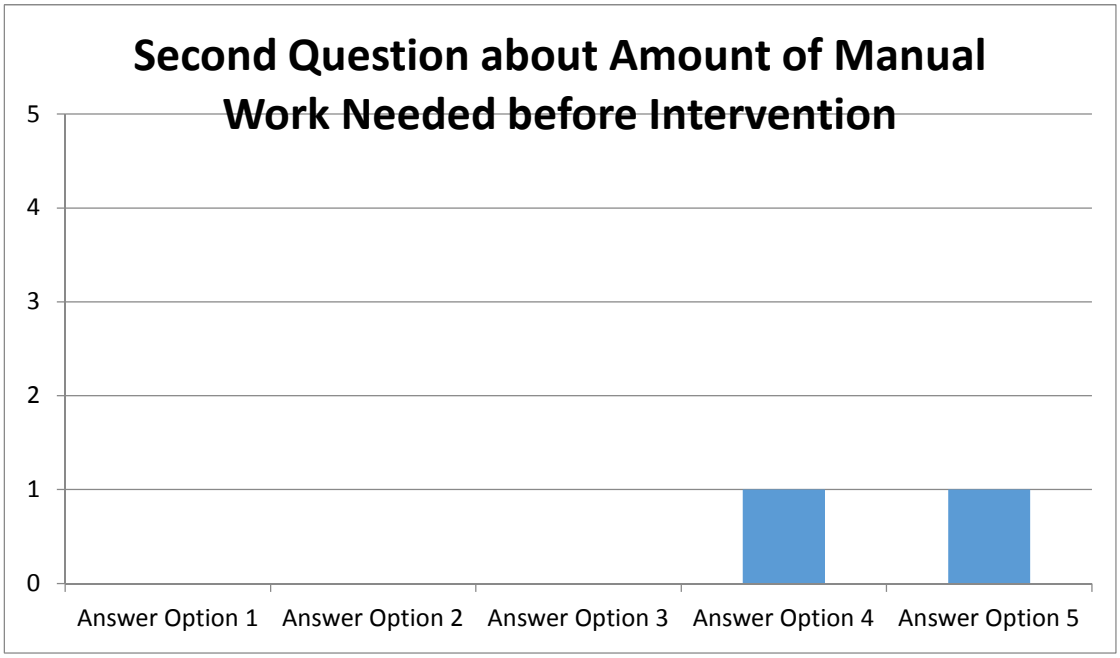
ent is with the current process of analyzing AFL's fuzzing results. Answer option one means really satisfied, answer option two means somewhat satisfied, answer option three means not satisfied nor dissatisfied, answer option four means somewhat dissatisfied and answer option five means really dissatisfied. Both find the current process of analyzing AFL fuzzing results somewhat dissatisfied as shown in Table 2 below.

Table 2. How satisfied is the respondent with the current process before intervention?



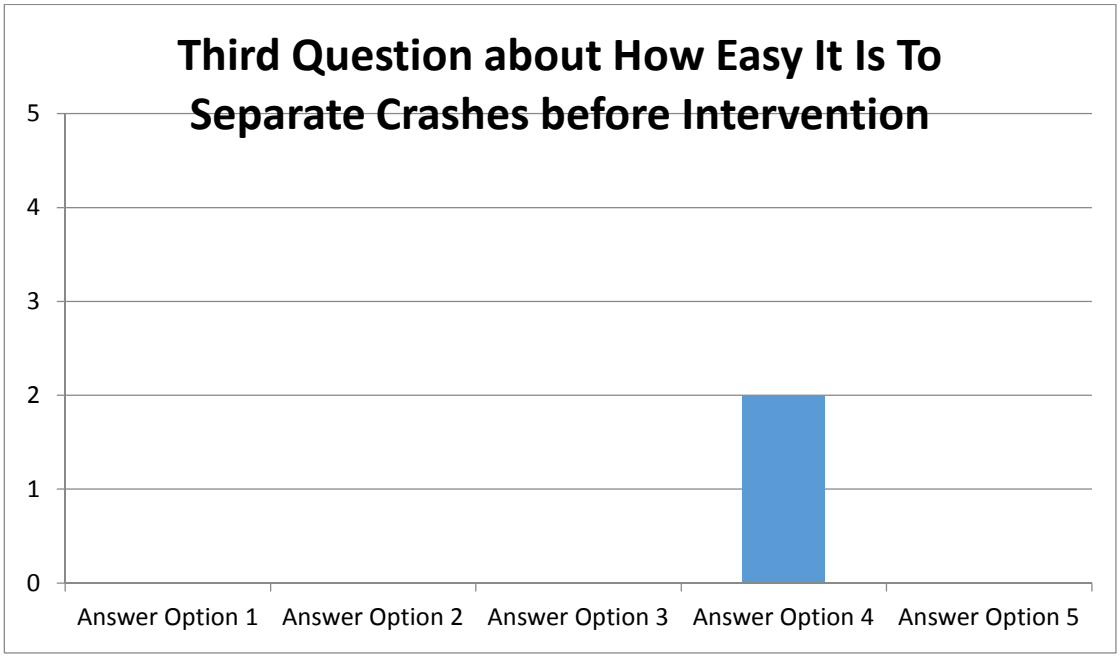
The second question focuses on evaluating the amount of manual work that is needed to do as a part of result analysis before intervention. The second question asks how the respondent sees the amount of manual work that has to be done as a part of analyzing AFL's fuzzing results. Answer option one means way too little, answer option two means could be more, answer option three means not too little nor too much, answer option four means could be less and answer option five means way too much. Both answers indicate that the amount of manual work could be less or is way too much as shown in Table 3 below.

Table 3. How does the respondent see the amount of manual work before intervention?



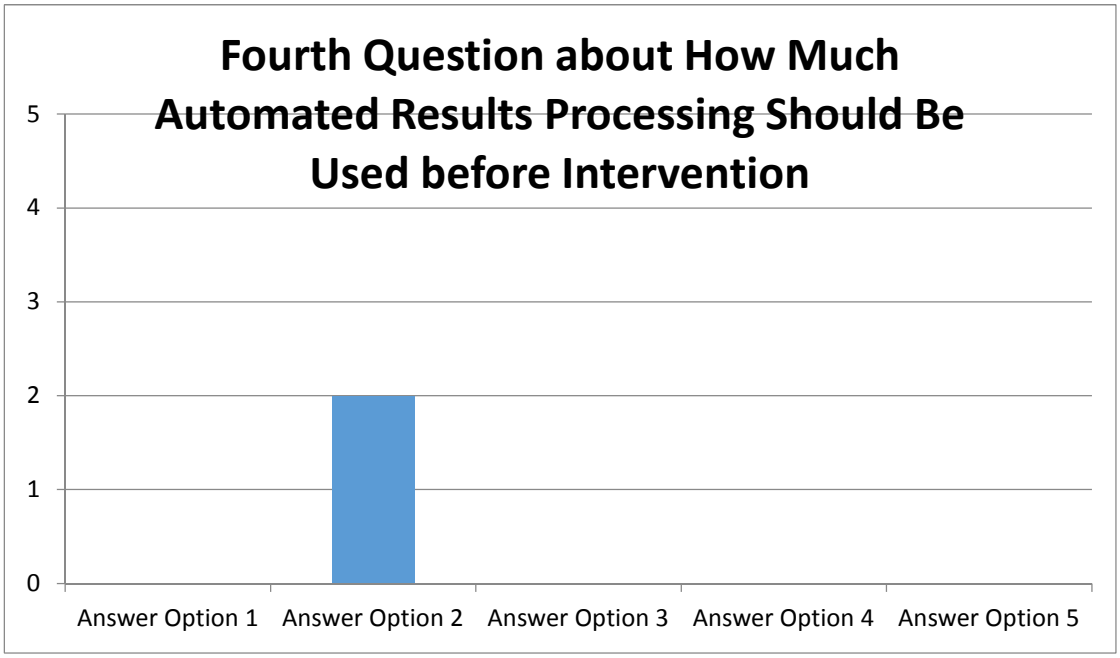
The third question focuses on measuring how easy it is to separate different crashes in the current process before intervention. The third question asks how easy it is in the respondent's opinion to separate different crashes. Answer option one means really easy, answer option two means somewhat easy, answer option three means not too easy nor too difficult, answer option four means somewhat difficult and answer option five means too difficult. Both answers indicate that separating different crashes is somewhat difficult as shown in Table 4 below.

Table 4. How easy it is to separate crashes before intervention?



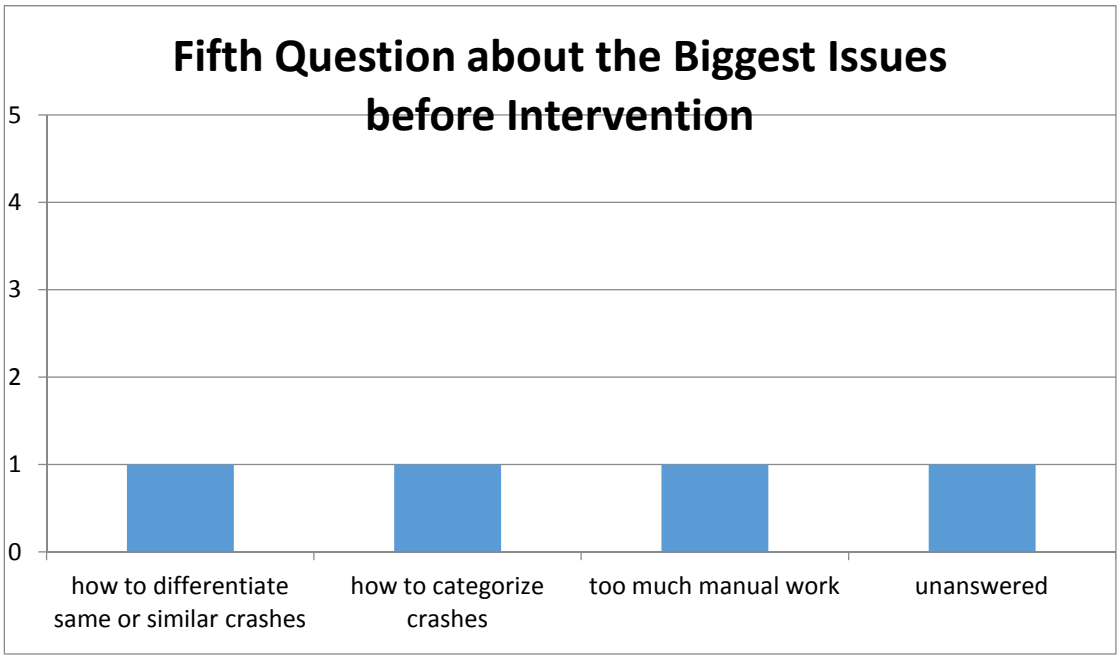
The fourth question focuses on evaluating how much automated result processing should be used before intervention. The purpose of question four is to find out should people be involved in crash analysis process and how provided program could be developed. The fourth question asks how much automated processing of results should be used. Answer option one means all should be automated, answer option two means automate as much as possible but let people check the results, answer option three means find the correct balance even though that may vary case by case, answer option four means automate some of the process but some of the work is left to the people, like running several commands and combining the results and answer option five means that the use of use of paper and pencil would be preferred. Both answers indicate that correct balance should be found even though that may vary case by case as shown in Table 5 below.

Table 5. How much automated results processing should be used before intervention?



The fifth question tries to find out two of the biggest issues currently in the result analysis process before intervention. The fifth open-ended question asks to name two biggest issues within the current analyzing process of AFL's fuzzing results. The themes are summarized in Table 6 and indicate difficulties in differentiating same or similar crashes, or how to categorize crashes and too much manual work is needed. However, one answer out of four was left unanswered so 25 percent of answers are summarized as unanswered.

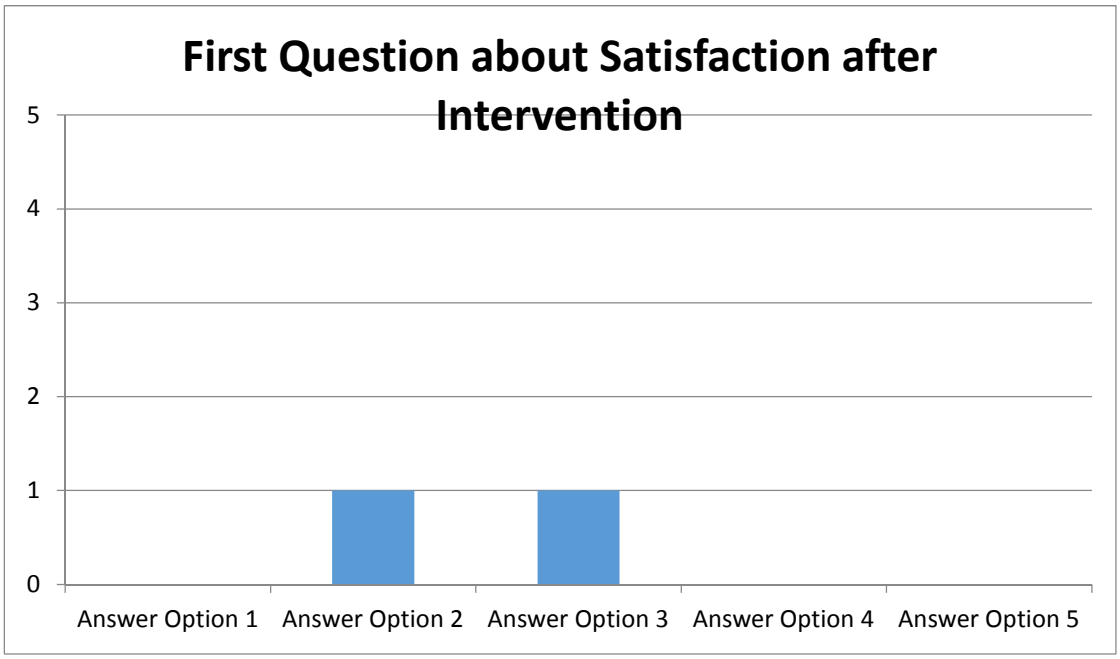
Table 6. The biggest issues in crash analysis before intervention.



4.2 Crash Uniqueness Questionnaire Results after Intervention

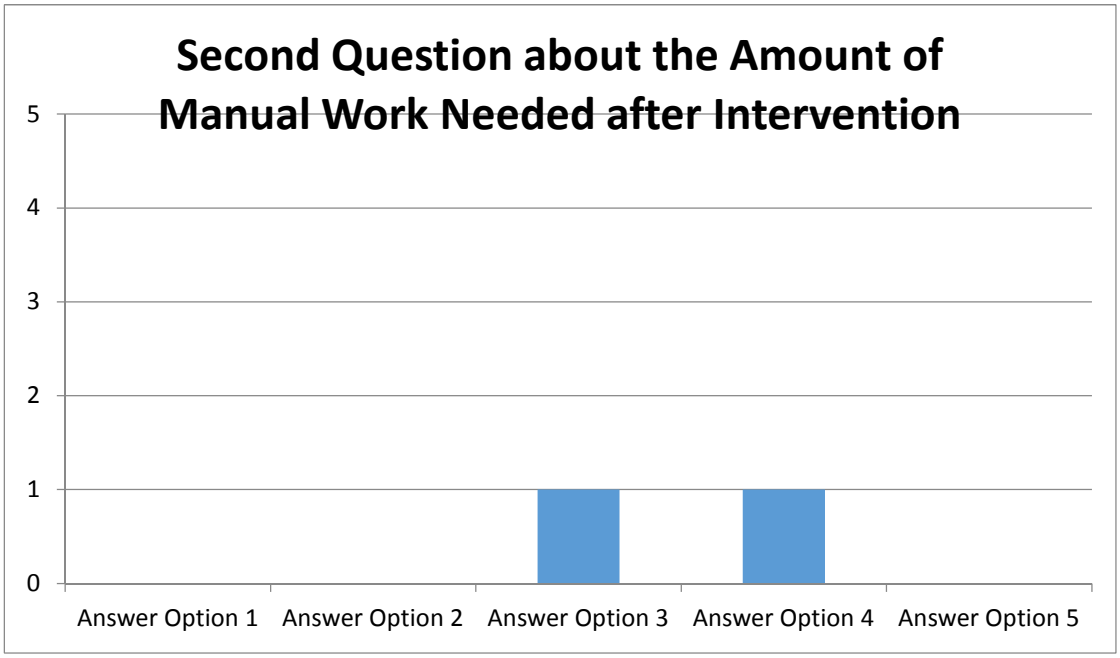
After intervention, the first question focuses on satisfaction with the Python program. The first question asks how satisfied the respondent is after the intervention about the current process of analyzing AFL's fuzzing results when a new program is used. Answer option one means really satisfied, answer option two means somewhat satisfied, answer option three means not satisfied nor dissatisfied, answer option four means somewhat dissatisfied and answer option five means really dissatisfied. Both find the satisfaction after intervention to be somewhat satisfied or either not satisfied nor dissatisfied as shown in Table 7 below.

Table 7. How satisfied is the respondent with the current process after intervention?



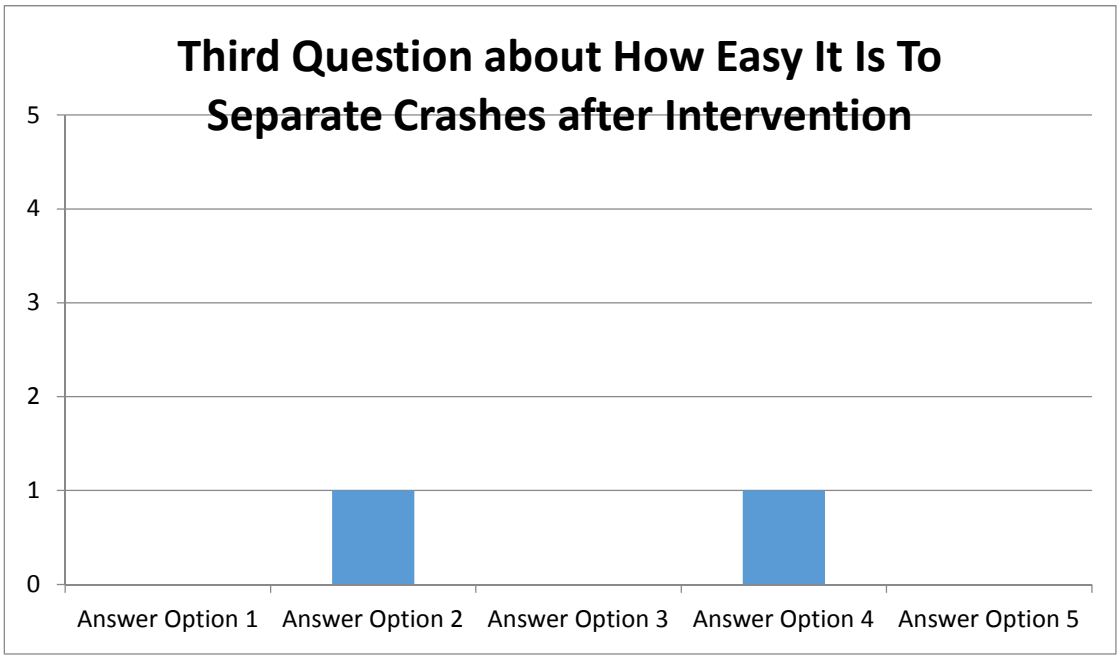
The second question focuses on evaluating the amount of manual work that is needed to do as a part of result analysis. The second question asks how the respondent sees the amount of manual work after the intervention that has to be done as a part of analyzing AFL's fuzzing results. Answer option one means way too little, answer option two means could be more, answer option three means not too little nor too much, answer option four means could be less and answer option five means way too much. Both answers indicate that the noticeable amount of manual work still needed as shown in Table 8 below.

Table 8. How does the respondent see the amount of manual work after intervention?



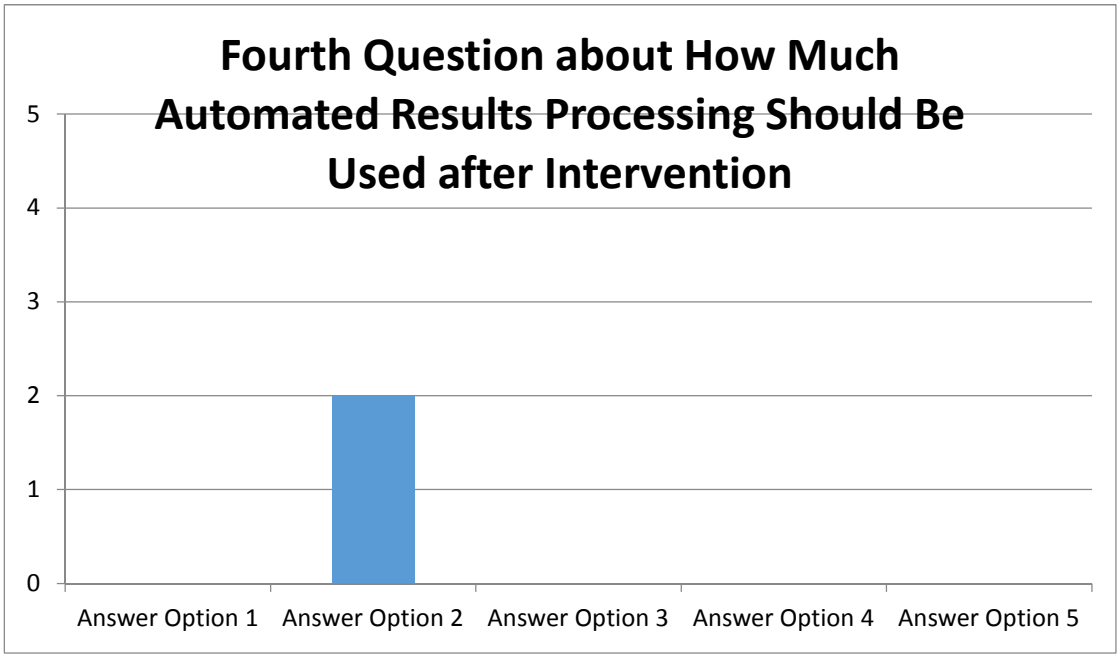
The third question focuses on measuring how easy it is to separate different crashes with the Python program. The third question asks after the intervention, how easy it is in the respondent's opinion to separate different crashes. Answer option one means really easy, answer option two means somewhat easy, answer option three means not too easy nor too difficult, answer option four means somewhat difficult and answer option five means too difficult. Both answers are split on the other ends of the scale as shown in Table 9 below. One finds that after intervention separation of different crashes is somewhat easy but the other finds that separation of different crashes is still somewhat difficult.

Table 9. How easy it is to separate crashes after intervention?



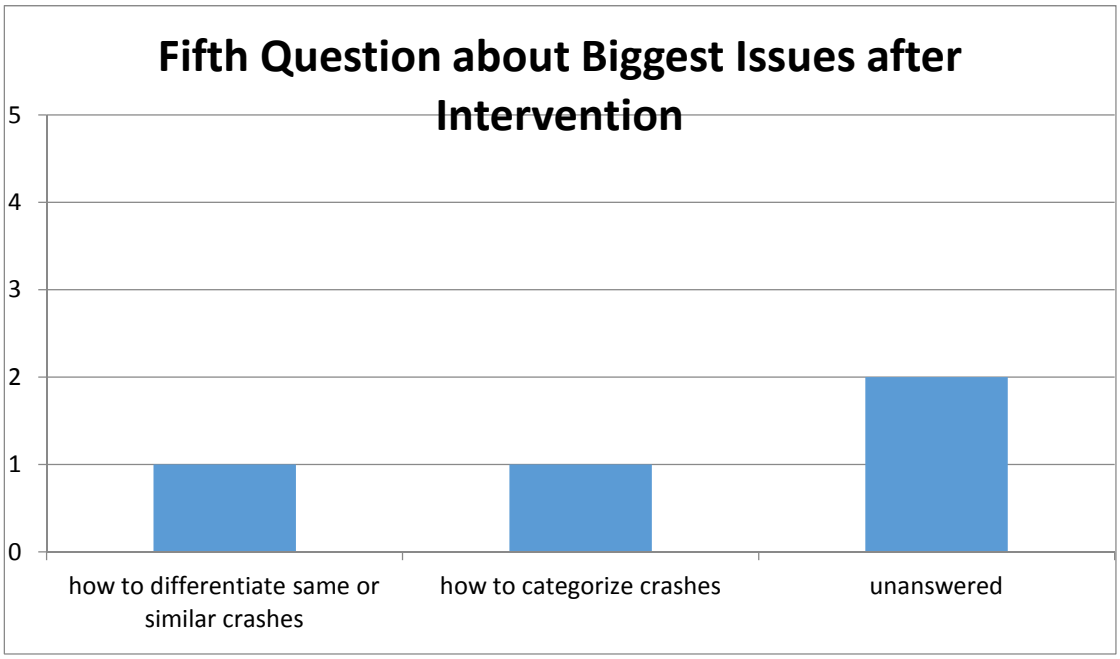
The fourth question focuses on evaluating how much automated result processing should still be used. The purpose of question four is to find out should people be involved in crash analysis process and how provided program could be developed. The fourth question asks after the intervention as a whole, how much automated processing of results should be used. Answer option one means all should be automated, answer option two means automate as much as possible but let people check the results, answer option three means find the correct balance even though that may vary case by case, answer option four means automate some of the process but some of the work is left to the people, like running several commands and combining the results and answer option five means that the use of use of paper and pencil would be preferred. Both answers indicate that correct balance should be found even though that may vary case by case as shown in Table 10 below.

Table 10. How much automated results processing should be used after intervention?



The fifth question tries to find out two of the biggest issues still lingering in the analysis process after intervention. The fifth open-ended question asks to name two biggest issues within the analyzing AFL's fuzzing results. Themes are summarized in Table 11 and indicate difficulties in differentiating same or similar crashes, or how to categorize crashes and too much manual work is needed. However, two answers out of four were left unanswered so 50 percent of answers are summarized as unanswered.

Table 11. The biggest issues in crash analysis after intervention.



4.3 Crash Uniqueness Intervention Impact

Did the intervention change anything? Kananen (2015, 58) asks that is it enough that at least something was changed? Table 12 shows that satisfaction results are moved from answer option four that means somewhat dissatisfied towards answer options two and three that mean somewhat satisfied and not satisfied nor dissatisfied. Based on Table 12, satisfaction has been improved after the Python program is provided to JyvSectec.

Table 12. Satisfaction results before and after intervention.

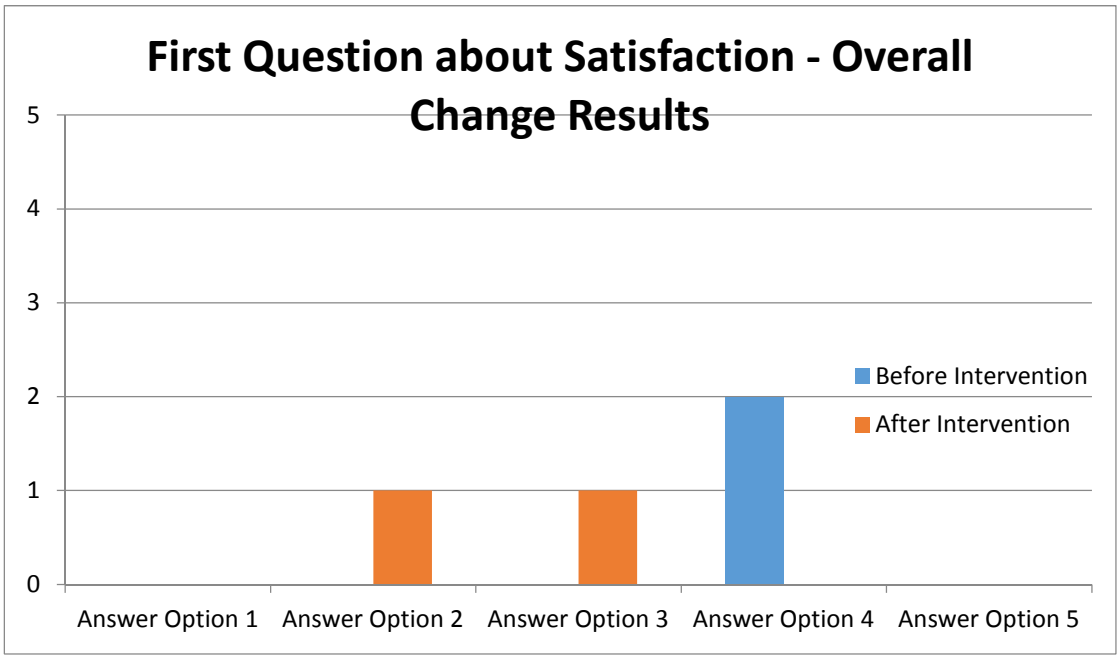


Table 13 shows that the amount of manual work that is needed to do as a part of result analysis is moved from answer options four and five one step to the left towards answer options three and four. Answer option four means that the amount of manual work could be less and answer option five indicates that amount of manual work is way too much. Answer option three indicates that the amount of manual work is not too little nor too much. Based on Table 13, amount of manual work needed to do as a part of result analysis is demised after Python program is provided to JyvSectec but there is still a noticeable amount of manual work needed.

Table 13. Amount of Manual work results before and after intervention.

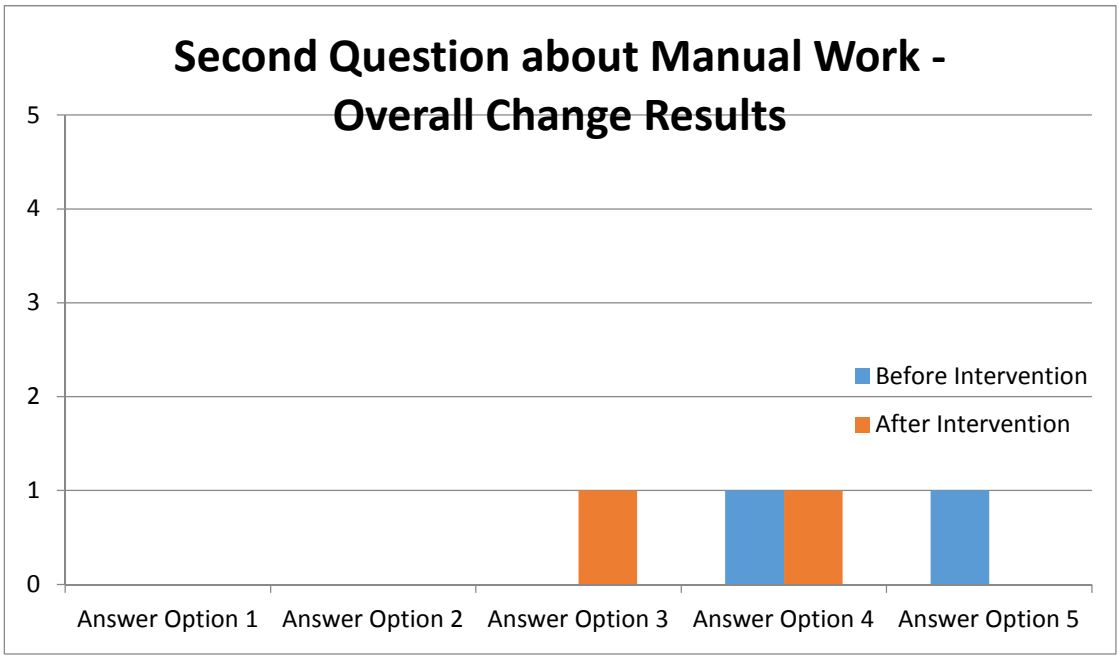


Table 14 shows mixed results on how easy it is to separate different crashes. Before intervention, the answer option four prevailed and answer option four indicates that separating different crashes is somewhat difficult. One of the answers after intervention shows that separating different crashes is somewhat easy but the other answer indicates that separating different crashes is still somewhat difficult. Based on Table 14, separating different crashes still needs plenty of work but crash analysis that JyvSectec does with SQLite3 is out of the scope of this thesis.

Table 14. Results of how easy it is to separate different crashes before and after intervention.

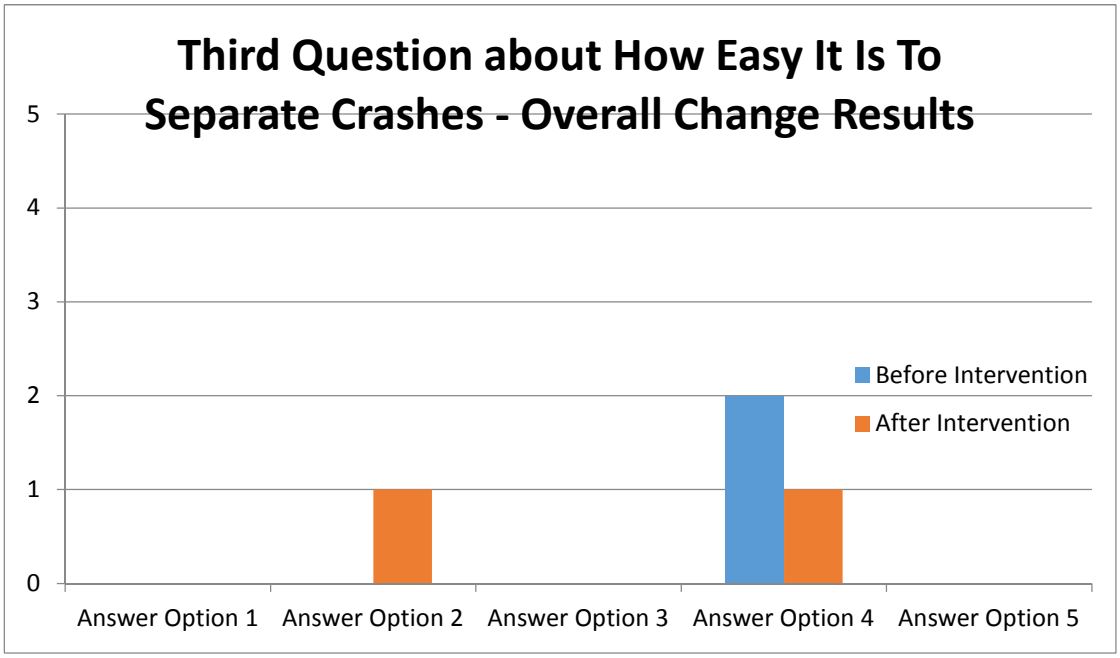


Table 15 shows no change when it comes to automated result processing that should be used. Answer option two means as much as possible should be automated but people should check results. The question points out the need for humans in the crash analysis process but also provides an option to develop the provided Python program to provide, for instance, some user interface for SQLite3. Based on Table 15, the crash analysis cannot be wholly automated.

Table 15. Results of how much automated results processing should be done before and after intervention.

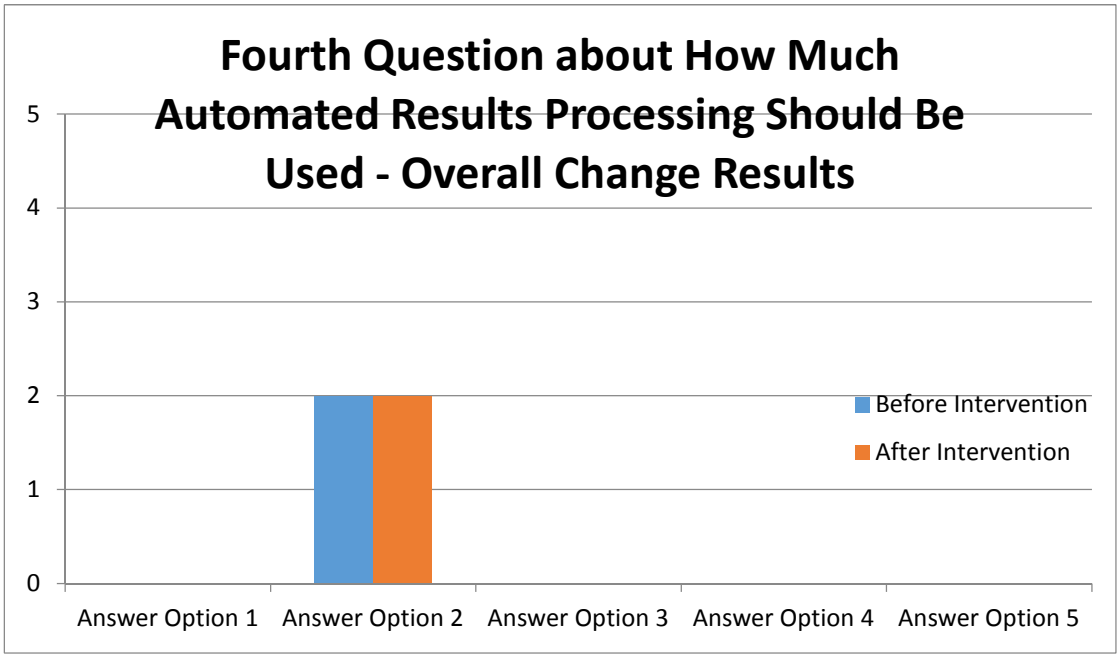
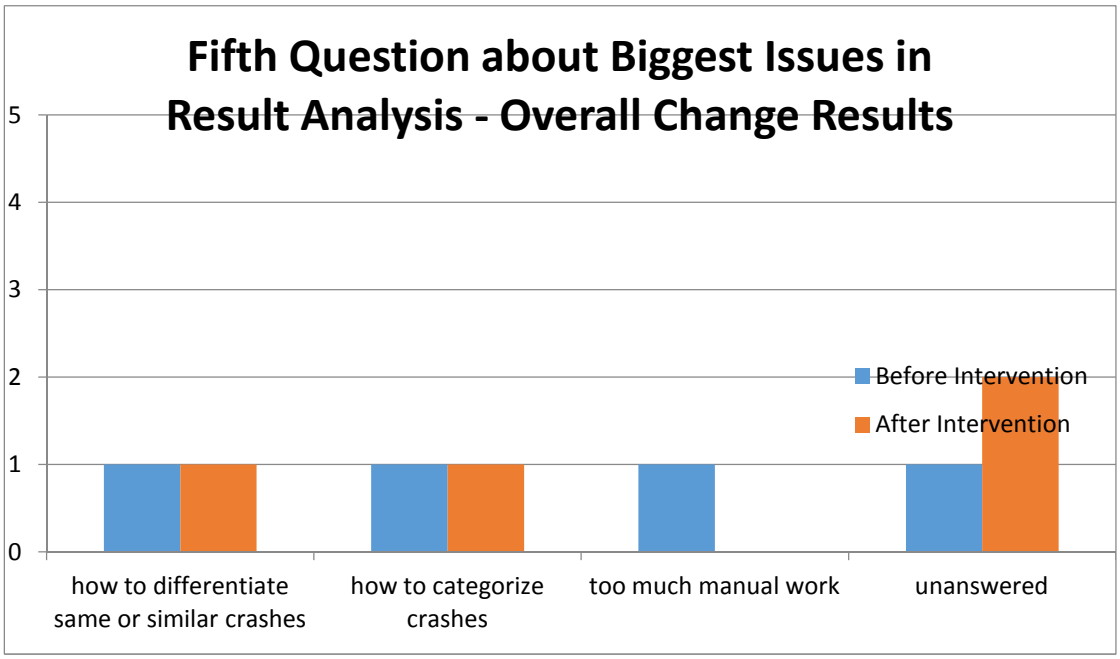


Table 16 shows that the biggest issues with crash result analysis still are the same before and after the intervention. Also the category unanswered is high. Based on Table 16 there is still a lot of work to do to make differentiating same or similar crashes easier and how to categorize crashes.

Table 16. Results of theme categories about the biggest issues before and after intervention.



Answers to the main research questions are mixed. Based on the results of the questions one, two and three in Tables 12 to 14, the satisfaction is improved after the intervention, the amount of manual work is demised and separating different crashes has slightly changed. However, question three results show that separating different crashes still needs a great amount of work and one of the answers indicate that there is no change after intervention. Based on the intervention results, the current process that requires a great amount of manual work has been improved. However, the results of questions five in Table 16 confirms the difficulties in result analysis and highlights that differentiating same or similar crashes and crash categorization is not changed by intervention. One possible reason might be the fact that only a part of the crash analyzing process is targeted, and the crash analysis done by SQLite3 in JyvSectec is omitted from this design research. The results of question four in Table 15 point out that as much as possible should be automated, howver, humans should still check the results.

5 Discussion

This chapter examines the results of this design research from overall theoretical basis. Possible follow-up research areas are also pointed out in addition of evaluating reliability and validity.

Design research framework depicted by Kananen (2015) is followed. Because design research is not a research method on its own a blended methodology is used (Kananen 2015, 33–34). Qualitative research focuses on single research subject and applies only to the cases that have been investigated (Kananen 2015, 35). Therefore, reflection on the theory basis is weak because not a single research paper concerning both fuzzing and crash uniqueness improving was found. Also due to qualitative approach with crash uniqueness and research questions focusing on JyvSectec's case, the generalization is weak. Hirsjärvi et al. (2013, 182) stress that generalizations should not be done in qualitative researches. Also Kananen (2015, 58) points out that both change and intervention generalization is quite often weak.

Reliability assessment is approached by the use of both reliability and validity (Kananen 2015, 112). Reliability refers to the fact that the results are stable and not

caused by randomness (Kananen 2015, 112). Validity refers to the fact that right variables are under research (Kananen 2015, 112).

One of the major challenges to the reliability is the small size of JyvSectec. Only two people make the target group for the questionnaires. Thus, the target group should be described as a biased selection or sampling (Coolican 2009, 110). Nevertheless, design research may use a blended approach and it became obvious really late that the amount of persons in JyvSectec was only two. Another factor that affected why questionnaires were chosen was the fact of scheduling challenges, distance studies and non disclosure agreements that stated that none of the meetings were allowed to be held inside JyvSectec's premises. However, in retrospective the use of interviews would have been a better approach. The use of interviews might also have been favored if the author's role had been another.

When it comes to the questionnaires Coolican (2009, 204) points out that when Likert's attitude scale is used decisions must be made how the mid-point is interpreted. Does the mid-point imply a neutral aspect or torn between both directions (Coolican 2009, 204)? The questionnaires interpret the mid-point as neutral. Also should the background questions still be asked even though the target group consisted only of two ICT experts in JyvSectec? Kananen (2015, 55) also points out that the research frame provides trustworthy information provided that correct measurements and target group are used. In case of this design research, the target group consisted only of two people, which had to be taken into account.

Kananen (2015, 52) states that the researcher may play the role of observer in design research, yet, that role came with some disadvantages. The main disadvantage is the lack of being involved in the action and not seeing the real use of crash analysis in fuzzing. That also dictated the use of questionnaires. Not being an active participant also meant that I could not see the whole process. The crash analysis continues with SQLite3 after GDB, which is out of the scope of this design research; however, still an integral part of crash analysis and optimizing only one part of the process may not be sufficient as can be seen from results. Tables 14, 15and 16 indicate clearly that intervention is not about solving issues with separating different, same or similar crashes and how to categorize crashes.

Another factor that may impact reliability is the randomness in AFL's fuzzing. When the simple speeding program is fuzzed, the used values by AFL vary slightly between different runs. As demonstrated in Chapter 2 if simple fuzzing is used without instrumentation, only a small part of program paths might be reached. But is that simple speeding program representative of real world programs or too simple to draw conclusions? Because of this issue, lib-jpeg is also fuzzed and the crash dumps are processed with the developed Python program. The results in Appendix 7 show that the same issue that JyvSectec had is reproduced.

The intervention part brings its own challenges and Kananen (2015, 58) asks if it is enough that something was changed in the intervention. Kananen (2015, 55) also stresses that the research frame does not ensure the fact that change is caused fully or partly by the intervention because third party factors may cause the change fully or partly.

Some experiments were performed with Libjpeg 6b version to see if the developed crash analysis program could be used with another program. AFL was used to fuzz libjpeg 6b with and without instrumentation. Un-instrumented fuzzing resulted two crashes in two hours but instrumented fuzzing revealed eleven crashes in nine minutes. Results of libjpeg 6b fuzzing are attached in appendices and AFL reported eleven unique crashes after fuzzing was stopped. A manual crash analysis after the developed Python program was run might indicate that out of those eleven crashes only two are unique showing that AFL reports several unique crashes that in manual crash analysis are not unique. That could be one of the interests in the follow up research. The use of SQLite3 after crash dumps have been produced would be another area of interest in the follow up research and might also answer how the whole process is affected by the use of the Python program. Now the crash analysis process is only approached from what kind of crash dumps are created and the analysis of crash dumps is left out of the scope of this research. But crash analysis is still a process and the intervention results point out that there exist several challenges after Python program is provided. The whole crash analysis could even be combined to the same program so that could be a user interface present to allow crash analysis work to be done with queries to the CSV database.

References

Arcuri, A., Zohaib, M., & Briand, L. 2012. *Random Testing: Theoretical Results and Practical Implications*. Accessed on 28 December 2016. Retrieved from IEEE.

Bath, G., & McKay, J. 2014. *The Software Test Engineer's Handbook. A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates 2012*. Rocky Nook Inc.

Bhat, A. 2015. The Significance of Testing throughtout the Software Development Life Cycle. *Internal Journal of Advance Foundation and Research in Science & Engineering.* Volume 1, Issue 9, February 2015. Accessed on 28 July 2016. Retrieved from http://www.ijafrse.org/Volume1/Vol_issue9/4.pdf

Black, R. 2007. *Pragmatic Software Testing – Becoming an Effective and Efficient Test Professional*. Wiley Publishing.

Cai, J., Yang, S., Men, J., & He, J. 2014. Automatic Software Vulnerability Detection Based on Guided Deep Fuzzing. *IEEE. Software Engineering and Service Science (ICSESS)*, 2014 5th IEEE International Conference. Accessed on 17.3.2016. Retrieved from IEEE.

Chess, B., & West, J. 2007. *Secure Programming with Static Analysis.* Pearson Education, Inc.

Copeland, L. 2003. *A Practitioner's Guide to Software Test Design*. STQE Publishing.

Coolican, H. 2009. *Research Methods and Statistics in Psychology*. Hodder Education.

DeMott, J. 2006. *The evolving art of fuzzing.* Accessed on 28 March 2016. Retrieved from http://vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf

DeMott, J., Enbody, R., & Punch, WF. 2007. *Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing*. BlackHat and Defcon 2007. Accessed on 28 March 2016. Retrieved from https://intelligentexploit.com/articles/Evolutionary-Fuzzing.pdf

Haikala, I., & Märijärvi, J. 1998. *Ohjelmistotuotanto*. Gummerus Kirjapaino Oy, Jyväskylä

Hirsjärvi, S., Remes, P., & Sajavaara, P. 2013. *Tutki ja kirjoita*. Bookwell Oy, Porvoo.

GCC, the GNU Compiler Collection. Accessed on 28 December 2016. Retrieved from https://gcc.gnu.org/

Godefroid, P., Kiezun, A., & Levin, M. *Grammar-based Whitebox Fuzzing.* Accessed on 3 April 2016. Retrieved from Citiseerx.

Godefroid, P., de Halleux, P., Nori, A V, Rajamani, S.K., & Schulte, W. 2008. Automating Software Testing Using Program Analysis. *IEEE Software 25.5* (Sep/Oct 2008): 30-37. Accessed on 28.3.2016. Retrieved from ABI.

Godefroid, P., Levin, M., & Molnar, D. 2012. SAGE: Whitebox Fuzzing for Security Testing*. ACM Vol. 55, no.3.* Accessed on 28 March 2016. Retrieved from ABI.

Kananen, J. 2015. *Kehittämistutkimuksen kirjoittamisen käytännön opas – miten kirjoitan kehittämistutkimuksen vaihe vaiheelta.* Tähtijulkaisut.

Khan, M. E., & Khan, F. 2014. Importance of Software Testing in Software Development Life Cycle. *IJCSI International Journal of Computer Science Issues*, Vol. 11, Issue 2. March 2014. Accessed on 6 January 2016. Retrieved from http://ijcsi.org/papers/IJCSI-11-2-2-120-123.pdf

Khan, ME., & Khan, F. 2012. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *Int. J. Adv. Comput. Sci. Appl,* 2012. Accessed on 28 March 2016. Retrieved from Citeseer

Libjpeg. 1998. *libjpeg.* Accessed on 26 February 2017. Retrieved from http://libjpeg.sourceforge.net/

Littlefair, T. CCCC - C and C++ Code Counter. Accessed on 27 July 2016. Retrieved from http://cccc.sourceforge.net/

Marcel, B., & Soumya, P. 2014. On the Efficiency of Automated Testing. *FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Pages 632–642. Accessed on 29 December 2016. Retrieved from ACM Digital Library.

Margaritelli, S. 2015. *Fuzzing with AFL-Fuzz, a* Practical Example (AFL vs Binutils). Accessed on 25 June 2016. Retrieved from https://www.evilsocket.net/2015/04/30/fuzzing-with-afl-fuzz-a-practical-example-afl-vs-binutils/

Mili, A., & Tchier. F. 2015. *Software Testing – Concepts and Operations. Wiley.*

JyvSectec. *Tietoa meistä.* Accessed on 6 January 2016. Retrieved from http://jyvsectec.fi/fi/tietoa-meista/

Kasurinen, J-P. 2013. *Ohjelmistotestauksen käsikirja*. Docendo.

Lagus, A. 2013. Ohjelmistohaavoittuvuus on kaikkien riesa. *Tietosuoja 2013:1*, 30–32. Accessed on 06 March 2016. Retrieved from https://www.tietosuoja-lehti.fi/index.php?mid=2&pid=32&aid=3047

McNamara, J. 2016. *XlsxWriter*. Accessed on 29 January 2017. Retrieved from https://xlsxwriter.readthedocs.io/index.html

Myers, G. J., Sandler, C., & Badgett, T. 2011. *Art of Software Testing (3rd Edition)*. John Wiley & Sons, 2011.

Oehlert, P. 2005. Violating assumptions with fuzzing. *IEEE Security & Privacy,* 3, 2, 58–62. Accessed on 21 March 2016. doi: 10.1109/MSP.2005.55 Refrieved from http://ieeexplore.ieee.org.ezproxy.jamk.fi:2048/stamp/stamp.jsp?tp=&arnumber=1423963

Pressman, R. S. 2000. *Software Engineering – A Practitioner's Approach. European Adaptation*. McGraw-Hill International (UK) Limited.

Python Software Foundation. *xlrd 1.0.0*. Accessed on 4 February 2017. Retrieved from https://pypi.python.org/pypi/xlrd

Samaroo, A. 2010. Life Cycles. In B. Hambling (Eds.), *Software Testing. An ISTQB-ISEB Foundation Guide. Second Edition.* British Informatics Society Limited, 34–55.

Shapiro, R., Bratus, S., Rogers, E., & Smith, S. 2011. *Identifying vulnerabilities in SCADA systems via fuzz-testing.* Critical Infrastructure Protection. Accessed on 2 April 2016. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-24864-1_5

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Yan, S., Kruegel. C., & Vigna G. 2016. *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*. Accessed on 2 April 2016. Retrieved from https://www.internetsociety.org/sites/default/files/blogs-media/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

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

Zalewski, M. 2016. *American Fuzzy Lop*. Accessed on 25 July 2016. Retrieved from http://lcamtuf.coredump.cx/afl/

Appendices

Appendix 1 First appendix

Simple C program fuzzed by AFL

The source code listing of simple C program that was developed to be fuzzed by AFL is listed below.

```
#include <stdio.h>
#include <stdlib.h>
#include "afl_test3.h"

#define SPEED 30

int main()
{
           char drivers_speed[3];
           int converted_drivers_speed = 0;
        char toJail = 'n';
           int pointsToBeAddedToTheLicence = 0;

           printf("Now, the speed limit here is %i.\n",SPEED);
           printf("Now, be honest, how fast were you speeding?");
           //gets(drivers_speed);
        converted_drivers_speed =  LueKokonaisluku();

        if (converted_drivers_speed <SPEED) {
           printf("Do nothing, keep eating a jelly donut...");
        }
        else if (converted_drivers_speed <=SPEED+5) {
           printf("Issue a warning and slow down a bit!");
        readComments();
```

→

```
            return (0);
        }
       else {
            printf("Really? I clocked you doing' %i. That's way over
%i\n", converted_drivers_speed, SPEED);
            printf("In fact, it's %i over speed limit\n", convert-
ed_drivers_speed-SPEED);
            printf("Didn't you see that %i MPH sign?\n",SPEED);
            if ((converted_drivers_speed-SPEED)<=15) {
                      //only add points to the drivers licence)
                       pointsToBeAddedToTheLicence =
ReadAmountOfPointsToBeAddedToLicence();
                   readComments();
                      return (0);
            } else {
                   printf("Should you be apprehended? Say N or else
you go to the prison\n");
                   toJail=lueKirjain();
                   readComments();
                      //pointsToBeAddedToTheLicence =
ReadAmountOfPointsToBeAddedToLicence();
                      return (0);
            }
        }
}

int LueKokonaisluku(void) {
 int ok=0;
 int arvo=0;
 char unwanterCharacters[40];
```

```
 unwanterCharacters[0] = 0;


 while (ok!=1) {
 fflush(stdin);
 if (scanf("%i", &arvo)==0 || arvo<0 || arvo>99) {
  gets(unwanterCharacters);
  printf("\nInvalid speed!\n");
  fflush(stdin);
  ok=0;
  exit(0);
  continue;
 } else
   ok=1;
  fflush(stdin);
 }
 return (arvo);
}


int lueKirjain(void) {
 int ok=0;
 int arvo=0;
 char toJail;
 char unwanterCharacters[40];
 unwanterCharacters[0] = 0;


 //while (ok!=1) {
 gets(unwanterCharacters);
 fflush (stdin);
 toJail = getchar();
```

```
 printf("\ntoJail on: %c", toJail);
 gets(unwanterCharacters);
 fflush (stdin);
 if (toJail != 'N') {
 printf("\nSo, you chose to go to the prison for speeding!\n");
 fflush(stdin);
 ok=1;
} else {
 ok=0;
 printf("\nSo, you chose not to go to the prison for speeding!");
 printf("\nDrive slower!");
 readComments();
 exit(1);
}
//}
 fflush(stdin);
 return (toJail);
}

void readComments(void) {
 int ok=0;
 int arvo=0;
 char noComments;
 char unwanterCharacters[40];
 char policeComments[250];
 char clientComments[250];
 unwanterCharacters[0] = 0;

 //gets(unwanterCharacters);
```

```
 fflush (stdin);
 printf("\nAdd both client's and police comments. Press N for no comments or any other for adding of comments.\n");
 noComments = getchar();
 gets(unwanterCharacters);
 fflush (stdin);
 if (noComments == 'N') {
 printf("\nNo comments added.");
 fflush(stdin);
 ok=1;
} else {
 ok=0;
 printf("\nAdd client's comments, 250 characters is the limit!");
 gets(clientComments);
 //gets(unwanterCharacters);
 fflush (stdin);
 printf("\nAdd police's comments, 250 characters is the limit!");
 gets(policeComments);
// gets(unwanterCharacters);
 fflush (stdin);
 }

printf("\nAll needed information is gathered, filing a report.\n");
 exit(1);
}

int ReadAmountOfPointsToBeAddedToLicence(void) {
           int ok=0;
           int arvo=0;
```

```
            //char amountOfPointsToBeAddedToLicence;
            char amountOfPointsToBeAddedToLicence[3]; //gets
            int convertedAmountOfPointsToBeAddedToLicence = 0;
            char unwantedCharacters[40];
            unwantedCharacters[0] = 0;

            //gets(unwantedCharacters);
            //fflush(stdin);
            printf("\nAdd number of points to be added!\n");
            //amountOfPointsToBeAddedToLicence = getchar();
        gets(amountOfPointsToBeAddedToLicence);
        printf("\namountOfPointsToBeAddedToLicence on: %c",
amountOfPointsToBeAddedToLicence);
            //gets(unwantedCharacters);
            //fflush(stdin);

            printf("\namountOfPointsToBeAddedToLicence on: %c",
amountOfPointsToBeAddedToLicence);

        //convertedAmountOfPointsToBeAddedToLicence =
amountOfPointsToBeAddedToLicence;
        convertedAmountOfPointsToBeAddedToLicence =
atoi(amountOfPointsToBeAddedToLicence);
            printf("\nAdded %d of points to the client's driver's li-
cence!\n", convertedAmountOfPointsToBeAddedToLicence);
            fflush(stdin);

            return (arvo);
}
```

The header file listing of simple C program that was fuzzed by AFL is listed below.

```
int lueKokonaisluku(void);
int lueKirjain(void);
void readComments(void);
int ReadAmountOfPointsToBeAddedToLicence(void);
```

Appendix 2 Second appendix

Cyclomatic Complexity of the Appendice 1 program.

Cyclomatic complexity and some other statistics of the fuzzed program of Appendix 1 by CCCC. Following results are from CCCC.

This table shows measures over the project as a whole.

- NOM = Number of modules
  Number of non-trivial modules identified by the analyser. Non-trivial modules include all classes, and any other module for which member functions are identified.
- LOC = Lines of Code
  Number of non-blank, non-comment lines of source code counted by the analyser.
- COM = Lines of Comments
  Number of lines of comment identified by the analyser
- MVG = McCabe's Cyclomatic Complexity
  A measure of the decision complexity of the functions which make up the program.The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram. The analyser counts this by recording the number of distinct decision outcomes contained within each function, which yields a good approximation to the formally defined version of the measure.
- L_C = Lines of code per line of comment
  Indicates density of comments with respect to textual size of program
- M_C = Cyclomatic Complexity per line of comment
  Indicates density of comments with respect to logical complexity of program
- IF4 = Information Flow measure
  Measure of information flow between modules suggested by Henry and Kafura. The analyser makes an approximate count of this by counting inter-module couplings identified in the module interfaces.

Two variants on the information flow measure IF4 are also presented, one (IF4v) calculated using only relationships in the visible part of the module interface, and the other (IF4c) calculated using only those relationships which imply that changes to the client must be recompiled of the supplier's definition changes.

Metric	Tag	Overall	Per Module

→

Number of modules	NOM	1	
Lines of Code	LOC	55	55.000
McCabe's Cyclomatic Number	MVG	9	9.000
Lines of Comment	COM	8	8.000
LOC/COM	L_C	6.875	
MVG/COM	M_C	1.125	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

CCCC was developed by Tim Littlefair as part of a PhD research project. This project is now completed and descriptions of the findings can be accessed at http://www.chs.ecu.edu.au/~tlittlef.

User support for CCCC can be obtained by mailing the list cccc-users@lists.sourceforge.net.

Appendix 3 Third appendice

Simple Python program for GDB log conversion

The source code for the Python program that converts AFL's crashes into GDB and outputs both xlsx and csv files with gdb crash dumps.

```
import argparse

import subprocess

import csv

import os

import sys

import shutil

import xlsxwriter

import codecs

import re


reload(sys)

sys.setdefaultencoding('utf8')


global_keep_separate_output_files = False

global_input_is_directory = False


import xlrd

import csv


"""

    Simple function to convert xlsx to csv used by this program
internally.

"""

def csv_from_excel():

          wb = xlrd.open_workbook('Crashes.xlsx')
```

→

```
            sh = wb.sheet_by_name('Sheet1')
            your_csv_file = open('Crashes.csv', 'wb')
            wr = csv.writer(your_csv_file, quoting=csv.QUOTE_ALL)

            for rownum in xrange(sh.nrows):
                wr.writerow(sh.row_values(rownum))

            your_csv_file.close()

"""
    Simple function to verify args.
                    - Checks that fuzzed program, input directory
and output directories exists.
"""
def Validate_args(args):
            print ('\n')
            print ('Checking args')

parser = argparse.ArgumentParser(description='Processes American
Fuzzy Lop logs with GDB and creates a  database of the results.')
parser.add_argument('fuzzed', metavar='program',
                    help='the fuzzed program')
parser.add_argument('input', metavar='input',
                    help='input directory of crashes')
parser.add_argument('output', metavar='output',
                    help='output directory to store analyzed crashes')
parser.add_argument('keep', metavar='keep',
                    help='if set to True keeps separate output files
instead of deleting them on exit.')
args = parser.parse_args()
```

```
#print(args)


if (os.path.lexists(args.fuzzed) == False):
           print('The fuzzed program could not be found')
           sys.exit()


if (os.path.lexists(args.input) == False):
    print('The input directory could not be found')
    sys.exit()


if (os.path.isdir(args.input) == True):
           print(os.listdir(args.input))
           list = os.listdir(args.input)
           number_files = len(list)
           print number_files
           global_input_is_directory = True


if (os.path.lexists(args.output) == False):
           print('The AFL\'s output file or directory could not be
found')
           sys.exit()


if (os.path.isdir(args.output) == False):
           print('The AFL\'s output is not directory')
           sys.exit()


if(len(os.listdir(args.output)) >0):
           print('The AFL\'s output directory still has some files!
Remove those.')
```

```
            sys.exit()


keep_files = args.keep[0]

keep_files = keep_files.upper()


if (keep_files == 'Y'):

                        print('Leaving separate output files')

                        global_keep_separate_output_files = True

else:

                        print('Not keeping separate output files')


print ('\n')


Validate_args(args)


print ('\n')


#Reads the number of files in the input directory

list = os.listdir(args.input)

number_files = len(list)

number_files = int(number_files)

path_to_input_files = os.getcwd() + '\\' + args.input + '\\'


#Copies the given fuzzed program to the same directory as the input
files in order to enable GDB to process it in batch run.

shutil.copy(args.fuzzed, args.input + '/' + args.fuzzed)


#Limits the files to be processed by GDB to the files that start with
id_

inputFilesList = [x for x in list if x.startswith('id:')]
```

```
#Processes every file in inputFilesList and runs GDB as a shell
command for every input files

amountOfFiles = len(inputFilesList)

prevdir = os.getcwd()


for i in range(amountOfFiles):

           ifile = open(args.input + '//run_gdb.bat', "wb+")

        os.chmod(args.input + '//run_gdb.bat', 0o777)

           filename=inputFilesList[i]

           #print(inputFilesList[i])

        parameters2 = "gdb --batch -ex 'r <"+filename+"'
"+args.fuzzed+" -ex bt >output_"+filename+".txt"

           ifile.write(parameters2)

           ifile.close()


           print ('\nSending file: ' + filename + ' to GDB')

           os.chdir(args.input)

           subprocess.call('./run_gdb.bat', shell=True)

           os.chdir(prevdir)


#print parameters2


#Before writing to a database file every output file from GDB is
moved to output directory

source =os.getcwd() + '/' + args.input

destination =  os.getcwd() + '/' + args.output

files = os.listdir(source)


prevdir = os.getcwd()
```

```
os.chdir(args.input)


for f in files:
    if (f.endswith("txt")):
        shutil.move(f, destination)


os.chdir(prevdir)


list = os.listdir(args.output)
number_files = len(list)
number_files = int(number_files)
path_to_output_files = os.getcwd() + '/' + args.output + '/'


# A workbook is created and a worksheet is added.
workbook = xlsxwriter.Workbook('Crashes.xlsx')
worksheet = workbook.add_worksheet()


# Start from the first cell. Rows and columns are zero indexed.
row = 0
col = 0


print ('\nStarting to write output files:')
for i in range(len(list)):
           #ifile = open(path_to_output_files + list[i], "rb")
           ifile = codecs.open(path_to_output_files + list[i], "rb",
encoding='utf8', errors = 'ignore')
           print ("\tWriting output file: " + list[i])
           reader = ifile.readlines()
```

```
            #print reader
         x = i+1
            worksheet.write_string(row, col, 'crash dump #:' + str(x))
            row += 1
            for item in (reader):
                        print item
                    item = re.sub(r'\([^()]*\)', '', item)
                        #worksheet.write_string(row, col, item)
                        worksheet.write(row, col, item)
                        row += 1
                        worksheet.write_blank(row, col, item)

workbook.close()

numberOfInputFiles = str(len(inputFilesList))
numberOfOutputFiles = str(len(os.listdir(path_to_output_files)))

print ('\nProducing output files, please wait...')
csv_from_excel()
print "\n" + numberOfInputFiles +  " input files processed and " +
numberOfOutputFiles + " output files done!"

#clean up
if (keep_files == 'N'):
                        print('\nRemoving separate output files')
                        os.chdir(args.output)
                        files = os.listdir(path_to_output_files)
                        for f in files:
                            if (f.endswith("txt") and len(f) >1):
```

```
os.remove(f)
```

Appendix 4 Fourth appendix

Questionnaire about current issues with analyzing AFL's fuzzing results in Jyvsectec.

This is a semi-structured questionnaire about current issues with analyzing fuzzing results in Jyvsectec. The purpose of this questionnaire is to provide data for the author's design research about fuzzing and evaluate the solution.

The current process was described in an email to be as following: After AFL fuzzing is completed the crashes are analyzed manually with GDB debugger and the duplicate entries are removed with bash3 script that uses GDB to get crash dumb. Those crash dumps are stored on sqlite3 database where searches are manually done in order to analyze the causes of crashes.

1. **Overall, how satisfied are you with the current process of analyzing AFL's fuzzing results?**

1 = really satisfied	2 = somewhat satisfied	3 = not satisfied nor dissatisfied	4 = somewhat dissatisfied	5 = really dissatisfied
☐	☐	☐	☐	☐

2. **Overall, how do you see the amount of manual work that has to be done as a part of analyzing AFL's fuzzing results?**

1 = way too little	2 = could be more	3 = not too little nor too much	4 = could be less	5 = way too much
☐	☐	☐	☐	☐

3. **Overall, how easy it is in your opinion to separate different crashes?**

1 = really easy	2 = somewhat easy	3 = not too easy nor too difficult	4 = somewhat difficult	5 = too difficult
☐	☐	☐	☐	☐

4. **Overall, how much automated processing of results should be used?**

1 = automate all	2 = automate as much as possible but	3 = find the correct balance even	4 = automate some of the process but	5 = I'd prefer the use of paper and pencil

→

	let people check the results	though that may vary case by case	some of the work is left to the people, like running several commands and combining the results	
☐	☐	☐	☐	☐

5. If you had to name two biggest issues within the analyzing AFL's fuzzing results which would those be?

_____

_____

Appendix 5 Fifth appendix

Questionnaire about current issues with analyzing AFL's fuzzing results in Jyvsectec after intervention.

This is a semi-structured questionnaire about current issues with analyzing fuzzing results in Jyvsectec. The purpose of this questionnaire is to provide data for the author's design research about fuzzing and evaluate the solution.

The current process was described in an email to be as following: After AFL fuzzing is completed the crashes are analyzed manually with GDB debugger and the duplicate entries are removed with bash3 script that uses GDB to get crash dumb. Those crash dumps are stored on sqlite3 database where searches are manually done in order to analyze the causes of crashes. An intervention to the analyzing process was attempted by providing a new program for that and this questionnaire is related to that.

1. **How satisfied are you <u>after the intervention</u> with the current process of analyzing AFL's fuzzing results when a new program is used?**

1 = really satisfied	2 = somewhat satisfied	3 = not satisfied nor dissatisfied	4 = somewhat dissatisfied	5 = really dissatisfied
☐	☐	☐	☐	☐

2. **How do you see the amount of manual work <u>after the intervention</u> that has to be done as a part of analyzing AFL's fuzzing results?**

1 = way too little	2 = could be more	3 = not too little nor too much	4 = could be less	5 = way too much
☐	☐	☐	☐	☐

3. **<u>After the intervention</u>, how easy is it in your opinion to separate different crashes?**

1 = really easy	2 = somewhat easy	3 = not too easy nor too difficult	4 = somewhat difficult	5 = too difficult
☐	☐	☐	☐	☐

→

4. **<u>After the intervention</u> as a whole, how much automated processing of results should be used?**

1 = automate all	2 = automate as much as possible but let people check the results	3 = find the correct balance even though that may vary case by case	4 = automate some of the process but some of the work is left to the people, like running several commands and combining the results	5 = I'd prefer the use of paper and pencil
☐	☐	☐	☐	☐

5. **<u>After the intervention</u>, if you had to name two biggest issues within the analyzing AFL's fuzzing results which would those be?**

_____

_____

Appendix 6 Sixth appendix

Sample crash results after Python program has been run.

Below is a sample of crash results after the Python program has been run. The crash dump contains the data provided by GDB but Python program can be configured to use different parameters. If crash analysis is performed for those crash dumps they seem to be caused by the same issue that is called in different parts of the program. However, AFL detects these as separate crashes.

```
crash dump #:1
Now, the speed limit here is 30.
Now, be honest, how fast were you speeding?Really? I clocked you doing' 77. That's way over 30
In fact, it's 47 over speed limit
Didn't you see that 30 MPH sign?
Should you be apprehended? Say N or else you go to the prison

Program received signal SIGABRT, Aborted.
0xb7fdbbe0 in __kernel_vsyscall
#0  0xb7fdbbe0 in __kernel_vsyscall
#1  0xb7e35057 in __GI_raise  at ../sysdeps/unix/sysv/linux/raise.c:55
#2  0xb7e36699 in __GI_abort  at abort.c:89
#3  0xb7e7319e in __libc_message  at ../sysdeps/posix/libc_fatal.c:175
#4  0xb7f03cb8 in __GI___fortify_fail  at fortify_fail.c:38
#5  0xb7f01e3a in __GI___chk_fail  at chk_fail.c:28
#6  0xb7f01dd1 in __gets_chk  at gets_chk.c:67
#7  0x08048de3 in gets  at /usr/include/i386-linux-gnu/bits/stdio2.h:236
#8  lueKirjain  at afl_test3.c:76
#9  0x0804884a in main  at afl_test3.c:38
crash dump #:2
Now, the speed limit here is 30.
Now, be honest, how fast were you speeding?
Program received signal SIGABRT, Aborted.
0xb7fdbbe0 in __kernel_vsyscall
#0  0xb7fdbbe0 in __kernel_vsyscall
#1  0xb7e35057 in __GI_raise  at ../sysdeps/unix/sysv/linux/raise.c:55
#2  0xb7e36699 in __GI_abort  at abort.c:89
#3  0xb7e7319e in __libc_message  at ../sysdeps/posix/libc_fatal.c:175
#4  0xb7f03cb8 in __GI___fortify_fail  at fortify_fail.c:38
#5  0xb7f01e3a in __GI___chk_fail  at chk_fail.c:28
#6  0xb7f01dd1 in __gets_chk  at gets_chk.c:67
#7  0x08048b5b in gets  at /usr/include/i386-linux-gnu/bits/stdio2.h:236
#8  LueKokonaisluku  at afl_test3.c:55
#9  0x08048713 in main  at afl_test3.c:17
crash dump #:3
Now, the speed limit here is 30.
```

→

Now, be honest, how fast were you speeding?Really? I clocked you doing' 40. That's way over 30
In fact, it's 10 over speed limit
Didn't you see that 30 MPH sign?

Add number of points to be added!

Program received signal SIGABRT, Aborted.
0xb7fdbbe0 in __kernel_vsyscall
#0 0xb7fdbbe0 in __kernel_vsyscall
#1 0xb7e35057 in __GI_raise at ../sysdeps/unix/sysv/linux/raise.c:55
#2 0xb7e36699 in __GI_abort at abort.c:89
#3 0xb7e7319e in __libc_message at ../sysdeps/posix/libc_fatal.c:175
#4 0xb7f03cb8 in __GI___fortify_fail at fortify_fail.c:38
#5 0xb7f01e3a in __GI___chk_fail at chk_fail.c:28
#6 0xb7f01dd1 in __gets_chk at gets_chk.c:67
#7 0x08048fc8 in gets at /usr/include/i386-linux-gnu/bits/stdio2.h:236
#8 ReadAmountOfPointsToBeAddedToLicence at afl_test3.c:146
#9 0x08048801 in main at afl_test3.c:33

Appendix 7. Seventh appendix

Sample crash results after the developed Python program has been run against Libjpeg-6b.

Below is a sample of crash results after the Python program has been run against Libjpeg-6b released in 1998. The crash dump contains the data provided by GDB but Python program can be configured to use different parameters. If crash analysis is performed for those crash dumps they seem to be caused by two crashes that are repeated. After a quick crash analysis is done, I would state that crash dumps one, two, three, five, eight, nine and ten are caused by the same issue. Crash dumps six, seven and eleven are also caused by the same issue but different from crashes mentioned above. However, AFL detects these as separate crashes. The total number of crashes was over 400 after nine minutes of fuzzing when source code is instrumented.

```
crash dump #:1

Program received signal SIGSEGV, Segmentation fault.
0x0804e352 in get_text_gray_row  at rdppm.c:152
152     *ptr++ = rescale[read_pbm_integer];
#0  0x0804e352 in get_text_gray_row  at rdppm.c:152
#1  0x080490e8 in main  at cjpeg.c:584
crash dump #:2

Program received signal SIGSEGV, Segmentation fault.
0x0804dfaa in get_text_gray_row  at rdppm.c:152
152     *ptr++ = rescale[read_pbm_integer];
#0  0x0804dfaa in get_text_gray_row  at rdppm.c:152
#1  0x080490e8 in main  at cjpeg.c:584
crash dump #:3

Program received signal SIGSEGV, Segmentation fault.
0x0804e27d in get_text_gray_row  at rdppm.c:152
152     *ptr++ = rescale[read_pbm_integer];
#0  0x0804e27d in get_text_gray_row  at rdppm.c:152
#1  0x080490e8 in main  at cjpeg.c:584
crash dump #:4

Program received signal SIGSEGV, Segmentation fault.
0x0804e352 in get_text_gray_row  at rdppm.c:152
152     *ptr++ = rescale[read_pbm_integer];
#0  0x0804e352 in get_text_gray_row  at rdppm.c:152
```

→

#1 0x080490e8 in main at cjpeg.c:584
crash dump #:5

Program received signal SIGSEGV, Segmentation fault.
0x0804e352 in get_text_gray_row at rdppm.c:152
152   *ptr++ = rescale[read_pbm_integer];
#0 0x0804e352 in get_text_gray_row at rdppm.c:152
#1 0x080490e8 in main at cjpeg.c:584
crash dump #:6

Program received signal SIGFPE, Arithmetic exception.
alloc_sarray at jmemmgr.c:406
406   ltemp = (MAX_ALLOC_CHUNK-SIZEOF) /
#0 alloc_sarray at jmemmgr.c:406
#1 0x08051000 in start_input_tga at rdtarga.c:437
#2 0x08049063 in main at cjpeg.c:568
crash dump #:7

Program received signal SIGFPE, Arithmetic exception.
alloc_sarray at jmemmgr.c:406
406   ltemp = (MAX_ALLOC_CHUNK-SIZEOF) /
#0 alloc_sarray at jmemmgr.c:406
#1 0x08051000 in start_input_tga at rdtarga.c:437
#2 0x08049063 in main at cjpeg.c:568
crash dump #:8

Program received signal SIGSEGV, Segmentation fault.
0x0804dfaa in get_text_gray_row at rdppm.c:152
152   *ptr++ = rescale[read_pbm_integer];
#0 0x0804dfaa in get_text_gray_row at rdppm.c:152
#1 0x080490e8 in main at cjpeg.c:584
crash dump #:9

Program received signal SIGSEGV, Segmentation fault.
0x0804e352 in get_text_gray_row at rdppm.c:152
152   *ptr++ = rescale[read_pbm_integer];
#0 0x0804e352 in get_text_gray_row at rdppm.c:152
#1 0x080490e8 in main at cjpeg.c:584
crash dump #:10

Program received signal SIGSEGV, Segmentation fault.
0x0804dfaa in get_text_gray_row at rdppm.c:152
152   *ptr++ = rescale[read_pbm_integer];
#0 0x0804dfaa in get_text_gray_row at rdppm.c:152
#1 0x080490e8 in main at cjpeg.c:584
crash dump #:11

Program received signal SIGFPE, Arithmetic exception.

```
alloc_sarray  at jmemmgr.c:406
406   ltemp = (MAX_ALLOC_CHUNK-SIZEOF) /
#0  alloc_sarray  at jmemmgr.c:406
#1  0x08051000 in start_input_tga  at rdtarga.c:437
#2  0x08049063 in main  at cjpeg.c:568
```