UiO : Department of Informatics
University of Oslo

Fuzzing analysis: Evaluation of properties for developing a feedback driven fuzzer tool

Master's thesis

Kris Gundersen

22/4 - 2014

1 Table of Contents

In this thesis fuzzing is defined as a method for discovering flaws in software by providing unexpected input and monitoring resulting errors or crashes. Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly referred to as fuzzers. If not otherwise stated, fuzzing will in this paper specifically denote file-format fuzzing. Fuzzing efficiency denotes how well the fuzzer is doing in the context of the number of crashes produced within a given time frame.

2 Abstract

Fuzzing, or fuzz testing is a technique that has been around for several years, and is now starting to become more widespread and commonly used during quality assurance of developed software. Even though fuzz testing typically is a highly automated technique, considerable time and effort is still required to completely test a program. A way to make the fuzzing process more efficient is therefore desirable, in an attempt to reduce the required time needed by only focusing on interesting parts, by making smart decisions. There are several aspects that may affect a fuzzing process and its efficiency, such as properties of the sample files used during file format fuzzing and the process used for acquiring the samples. When investigating the templates themselves it is typical to see that some parts of the file format is more important than other parts, and such information could potentially be very valuable when attempting to improve the fuzzing process.

This thesis demonstrates how to make use of feedback from instrumenting fuzzing executions to make smart decisions regarding what template to use as a basis for the remaining iterations of the fuzz process as a technique. This is done in an attempt to make fuzzing more efficient. Code coverage and trace differences will be used as metrics to evaluate the strength of a given mutated sample. In addition to these two metrics, unique crashes will be used in several occasions. Furthermore several different aspects of the templates and their impact on the effectiveness of the fuzzing process will be investigated. This includes investigating the difference between the two ways of producing samples, generating or creating, and recording how much impact one specific feature of the input format will have when either removed or added. Additionally, effort will be made to identify important sections of the input file format by seeing what parts of the file that are often executed and seeing if new parts are produced that originally did not exist, after mutating the file. Finally some research will be conducted to survey where in the trace a crash typically lies, to see if it is close or not to the normal trace, and thus if high code coverage is important.

Section I

3 Foreword

This thesis is written as a part of my degree “Master of Science in Informatics: Distributed Systems and Network” at the University of Oslo, Faculty of Mathematics and Natural Sciences, Department of Informatics. The thesis is written in collaboration with the Norwegian Defence Research Establishment (FFI) and UNIK University Graduate Center.

3.1 Acknowledgments

I want to thank FFI for letting me write a thesis about a fascinating and very relevant topic. In particular, I would like to thank my supervisor at FFI, Trond Arne Sørby, and my supervisor at the Department of Informatics, Audun Jøsang, for excellent guidance and advice. I would also like to thank Tormod Kalberg Sivertsen for participating in discussions and providing valuable feedback and suggestions

Kris Gundersen

22nd of April 2014

4 Introduction

4.1 Software security and quality assurance

Software security is the idea of engineering software so that it continues to function correctly under malicious attack. All too often, malicious intruders can hack into systems by exploiting software defects. By any measure, security holes in software are common, and the problem is growing[2]. Improving software security is therefore an important issue with existing, as well as with developed software, and effort should be made to improve it. The way to improve software security is to make sure no software defects are present during release of the software, or to identify and remove (fix) them. There are various techniques used to identify software defects.

Software quality assurance is a technique used when developing new software to ensure that the quality of the final product is as desired. The software quality assurance is performed throughout the entire project, from defining the requirements to testing and release. The specific methods that are used during the software quality assurance process may vary. One important step of the software quality assurance technique is the testing phase, to make sure that there is no software defects present before releasing the product. Software testing is a huge topic in itself and there are many different testing techniques, for instance static analysis or fuzzing.

4.2 Fuzzing and thesis theme introduction

Below, is a very brief description of the main concept of the fuzzing technique.

Fuzz testing, as the technique may also be called, attempts to automatically discover bugs[1] in the targeted program and can therefore be called a negative software testing method. Basically a fuzzing tool creates random and often corrupt input data, which is then fed to the program being fuzzed. The execution of the program in question is then monitored and if a crash occurs, ideally the input data which caused the crash, along with the location of the crash is recorded. Several iterations with different input data are then executed to reveal as many bugs as possible and to cover as much of the targeted code as possible.

In other words, the main purpose of performing fuzz testing on a piece of software is to uncover security critical flaws leading to denial of service, degradation of service, or other undesired behavior. When fuzzing, we are interested in finding any and all inputs that can

[1] Bugs are parts of a program which do not function as intended, and may either result in incorrect results or a program crash.

trigger undefined or insecure behavior. A security flaw detected after deployment of software is much more expensive to fix compared to detecting and fixing it pre-deployment and it is therefore important to find as many bugs as possible before releasing the software. Static analysis typically yields a rather high rate of false positives, but all bugs found with fuzzing are true positives[3].

The overall goal of this thesis is to investigate ways to improve the fuzzing process in relation to speed and results, both during fuzzing but also beforehand to improve the samples and other prerequisites used. The goal is to investigate and produce methods of performing these operations without having any prior knowledge of neither the target program nor the file format used. Another requirement is that the processes should be highly automatic with as little need for user interaction as possible. In addition there will be some focus on analysis of several of the fuzzing steps and extracting coherent statistics and information as well as uncovering key areas to focus to improve efficiency. As all work conducted in this project is made under the assumption that no prior knowledge about the target programs internal structure should be required, the results of the thesis will reflect what possibilities and options that are possible to achieve based on such terms.

4.3 Overview, Plan and research question(s)

4.3.1 Nature of the thesis

The tasks or objectives of this thesis were originally designed as a series of smaller tasks, which focus on various aspects of fuzzing. These aspects range from improvements which can be done, to gathering the optimal template set and which affects they have on the fuzzing process, to how one may use feedback from instrumentation to generate new fuzz cases. In other words there are several aims for this thesis. Among the most important is on an overall level to investigate various properties and aspects of the fuzzing process and see what effects they have on the results of a fuzzing run. The thesis also attempts to uncover which aspects that affect the fuzzing results the most, resulting in critical areas and parts to focus on when trying to optimize fuzzing. Finally the thesis attempts to find a way to use feedback from instrumentation to improve efficiency and fuzzing results. Due to the existence of several minor objectives, the research is divided into multiple standalone research tasks, and conclusions and analysis of each of these tasks are presented. These "modules" of objectives/tasks, will serve as the building blocks and will be used to answer the major research questions of this thesis.

The various research tasks are briefly described as follows:

Research area	Brief description
Code Coverage	How far from a trace does a bug actually occur? Investigate hypothesis: Normally lies close. Approach: Trace Analysis. Important for assessing the importance of good code coverage.
Pre-fuzz	Investigate the impact of various template properties on fuzzing results. (Measure differences in results, trace and code coverage for various features of a sample). Learn what parts of the data model that are interesting (which segments are seldom and commonly occurring in templates). Investigate the most rewarding way of producing samples, creating from a program or generating data from a data model.
Fuzzing	Use feedback from instrumentation to generate new fuzz cases. Is it worth it to instrument all fuzz cases(based on time required and improvement of fuzzing results) ?

Table 1: Research tasks

Each objective will be thoroughly discussed and explained when presenting the work performed and results gathered for the objective in question, in Section 7:The work process, results and analysis. The term research task will occasionally be used throughout the project.

4.3.2 Thesis structure

The thesis structure is composed of several sections with subsections, each having its individual purpose as to guide the reader through the thesis. In section one, the foreword is presented with some disclaimers and an “Abstract” section to briefly give an indication of the content of this paper. Following the foreword, is the introduction, meant as a section where the reader can get familiar with and eased into the general terms, techniques and concepts of the various topics touched by this paper, as well as being introduced to the research question, the plan and goal of the thesis. The Presentation serves as the last subsection of the first section containing a presentation of the target programs, metric to be used, the various tools utilized and important decisions.

Section two is one of the most important parts and consists of a presentation of the actual work conducted while performing the research and the practical parts of the thesis. This includes the procedures and techniques used, with adequate motivations, where required, for why the used processes are chosen and are correct. When needed some minor code snippets or similar resources are presented. This section will also contain the results of the work conducted, as well as appurtenant analyses, where applicable. The actual results may vary depending on the various research cases, and may range from concrete numbers or facts, to a discovery of the properties of some parts of the fuzzing process. This section is the main part of this paper as it contains the techniques and methods used for the various research tasks, serving both as an explanation of how the work was conducted, but also motivates and describes relevant information. In addition, as mentioned, this section also holds all the results which can be of high interest and may serve as a basis for readers to see connections and draw conclusions.

The third section holds the conclusion which can be drawn from the results deduced in the previous section. An answer to the main research questions will also be given partly based on the individual research cases conclusions and work. Finally some thoughts on the work process, the results and also on future work will be given. This section is also of great importance as this sums up the all the research performed in the project and also makes conclusions for each of the research questions.

The table of figures, table of tables and the bibliography compose the final section of this paper, solely followed by an appendix.

4.3.3 Research question(s)

As previously discussed, this thesis is mainly composed of several smaller research cases, but these will mainly serve to discover pieces of the major research question. The research questions for this thesis are:

RQ1: Does the templates used during file format fuzzing impact the fuzzing process and results significantly, and if so, how?

Explanation: This research question concerns the templates that are used in file fuzzing and how they are used during the fuzzing process. As the templates used in a program can determine what functionality in the target program that is executed, it is of great interest to see how various properties of the samples used can impact and affect the behavior and the executed code of the target program. This can in turn be used to optimize further fuzzing runs based on knowledge of typical effects of tweaking certain properties or aspects of the templates used.

RQ2 (Main RQ): Is it possible to improve the efficiency of a file format fuzzer by focusing on the template used based on feedback from instrumentation.

Explanation: This research question is concerned with finding a way to improve the results yielded from file format fuzzing, and partially builds on the first research question concerning the properties of the samples used. Interesting issues to shed light on is to see if it is possible to use feedback from current fuzzing iterations to gradually improve the samples used based on the feedback from executions with previously used samples. More specifically, is it possible to use the feedback to detect when a new sample is to be considered better than the one currently used, based on given criteria, and in turn utilize this information to improve the fuzzing process itself, in the context of yielded results.

By uncovering good answers to these questions it should be possible to perform file format fuzzing in a more optimized and rewarding way. It will also enable a broader understanding of the relation between certain aspects of file format fuzzing. The overall goal of finding good answers is to improve certain stages or aspects of the entire file format fuzzing process and all related elements.

5 Theory

5.1 Software security and quality assurance

The main motivation for software security is to avoid security incidents. A security incident can be an event where someone unauthorized is able to compromise the target system's security, thereby granting them access to the system and/or data through and active attack. Most commercial software today has some sort of security problem and most are therefore

considered immature from a security perspective. According to Takanen one fact has emerged from the security field: *Software will always have security problems*. Almost all software can be hacked[4].

As poor software security potentially can harm or damage the person or company relying on the software in case of an attack or other unforeseen disasters, the need for improving the security of software is therefore gradually increasing and many software development companies have designated people trying to assure the quality of developed software, for instance in terms of security. These quality assurance people perform various tasks, such as validation and verification of the software product. With the complexity of the programs produced today, it is almost impossible to have tests and test cases that cover every single part of the program. As we will see, fuzzing is one of the testing techniques that are more and more used in relation to software testing when performing quality assurance. As stated by Takanen, when looking at fuzzing from a quality assurance perspective, fuzzing is a branch of testing; testing is a branch of quality control; quality control is a branch of quality assurance

When working with quality assurance, program analysis may be used to ensure the quality of the software. When performing program analysis it is typical to investigate the program at a certain granularity. There are mainly three different granularity levels used, whereas the two first are the most commonly used:

1. Basic blocks - A block of code with a single entrance and a single exit point
2. Functions – a subroutine of a program with one entry and several exit points
3. Instructions – a single CPU instruction

Which granularity level that is desirable depends on the type of analysis to be performed, the context of the work and other factors such as the granularity used by other third party tools needed later or earlier in the work process.

5.2 Fuzzing

5.2.1 Concepts and purpose

Below is a rough explanation of the purpose of fuzzing. A more detailed description is given in the remaining parts of this section.

The aim of fuzzing is to uncover weaknesses and vulnerabilities in a program, and as some of which typically are critical security problem that can be exploited to attack the program and take control. Due to this, fuzzing is today a rather popular technique amongst both quality assurance engineers trying to secure their software, as well as by hackers trying to break some piece of software.

Fuzzing has one goal, and one goal only: to crash the system; to stimulate a multitude of inputs aimed to find any reliability or robustness flaws in the software. For the security people, the secondary goal is to analyze those found flaws for exploitability[4].

5.2.2 Techniques

Fuzzing may be used for various types of applications such as file fuzzing, Web browser fuzzing, ActiveX fuzzing or network fuzzing

In addition to variation in the target to fuzz, there are also several variations regarding the construction and how they work, especially with regard to how the fuzzed input values are created.

There are two major different approaches to fuzzing: Mutation-based and Generation-based [5]. Mutation based fuzzing uses a set of templates, which are valid input files for the targeted application, that are used to perform random mutations yielding new input files with some changed data to be fed by the fuzzer. The advantages of such an approach are that little knowledge or documentation of the input format is required, only sample file(s) for the given application is needed. Such fuzzers are normally fairly easy to reuse with other software targets.

Generation-based fuzzers generate the input from a specified model of the input format, and therefore a rather detailed knowledge of the input format is desirable to achieve good results with such fuzzers. Generation based fuzzers do not rely upon randomness but are strictly heuristic[6].This normally results in the advantage of having a smaller set of tests while yielding a greater number of bugs uncovered, compared to a mutation based fuzzer [5, 6]. However a generation based fuzzer will typically only work for other programs that expect the same input format as the initial target program.

Typically one also differentiates between black-box fuzzing and white-box fuzzing. When performing fuzzing without any information about neither the target binary nor the input format, this is referred to as black box fuzzing. Black box fuzzing is the most common fuzzing type as some of the strengths of fuzzing are that little or no information is needed about the target program to detect faults by an automated process. However, due to the lack of insight in the target program when using black box fuzzing on complex programs, the efficiency (how many crashes that are uncovered in a given time frame) of the fuzzer might be severely degraded. There are several reasons for possible degraded efficiency, for instance when fuzzing a program which validates the input directly after receiving it from an external source and exits if anomalies are found. Such integrity mechanisms will abort execution when run through a fuzzer. Work has been done on this specific area. Taintscope is an example of a fuzzing tool which tries to bypass integrity checks[1] . The following figure is

from the authors of Taintscope, describing the tools program flow:

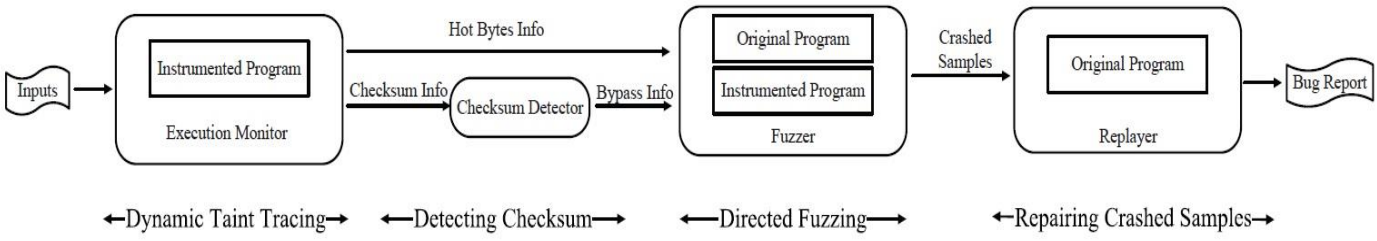


Figure 1: TaintScop System Overview [1]

Furthermore, black-box fuzzing might miss parts of the code space (low code coverage[2]) or be unable to detect a fault in some instances when the program consists of complex logic[7]. It is possible, and common, to perform dumb fuzzing in black box fuzzing cases, for instance by mutating an existing file.

White box fuzzing is somewhat similar to black-box fuzzing in the sense that there really is no need of any prior knowledge of the target application. White box fuzzing is different from black-box fuzzing as it extends its scope from unit testing to whole-program security testing[7]. A typical white-box fuzzer uses symbolic execution during dynamic testing to collect constraints on the inputs used in conditional branches. These constraints are then used to execute different code paths in the program by systematically negating the constraints before solving them with a constraints solver. To be able to reach as much of the program as possible, the described process is repeated using novel search techniques[7]. This should theoretically lead to a complete coverage of the target program, but due to certain limitations this is normally not the case during practical use. Limitations that may reduce the efficiency of a white box fuzzer are mainly caused by the fact that symbolic execution, constraint generation and constraint solving can be inaccurate as a result of complex programming logic such as pointer manipulation, external calls and library functions. Huge programs may also limit the usefulness due to the needed time to solve all constraints in an acceptable time frame. Due to the generated insight in where important branches are as well as a better understanding of the program flow, it is possible to achieve greater code coverage, and more quickly detect faults and identify vulnerabilities.

While discussing white box fuzzing, I would like to mention with box testing, as one term may be confused with the other. One could think that white box fuzzing was the opposite of black box fuzzing; White box fuzzing requires knowledge of the target program to a high extent. However as we have seen, this is not the case. White box fuzzing does not necessarily need any prior knowledge of the target program, but extracts information from

[2] How many percent of the program code that has been executed during execution of the program. It is common use number of functions or basic blocks as metrics. For finer granularity, the number of instructions may also be used.

the program during symbolic execution. White box testing on the other hand refers to test methods that rely on the internal structure of the software[8]. A white-box test approach can be used either to generate test cases or to measure the extent to which a given set of test cases covers all the code elements identified by the approach. Typically a white box test is constructed to work with one target program as the logic of a program will vary from one to another. The insight into the programs construction can for instance be acquired if source code is available.

5.2.3 History

The utilization of fuzzing in software testing has greatly increased since the testing technique was developed and emerged as a term in 1988[3] and is today a pretty common technique [5]. Originally the word “fuzzing” was just another name for random testing and was not really connected to or widely used in quality assurance of software.

Professor Barton Miller is considered by many to be the “father” of fuzzing as he and his Advanced Operating System Class developed and used a primitive fuzzer tool to test the robustness of their system and UNIX applications[9]. In 1995 the testing process was repeated with an extended set of UNIX utilities. At this point the fuzz testing simply consisted of passing random strings of characters to the target program. This is a rather primitive and pure black box approach, but the concept of fuzzing was at the time quite young[10].

During the period from 1988 (but primarily from 1999) to 2001 research was conducted focusing on a new model-based testing technique and other fuzzing techniques which could be used to automated the testing process of developed software[3]. The research mainly focused on network analysis and testing, and the approach was a mix of white and black box fuzzing. This marked an important milestone in the evolution of fuzzing as this approach yielded a larger number of faults than earlier achieved. The research was conducted in the PROTOS project at the University of Oulu. The goal of this research was to make companies developing software able to automatically discover security critical problems in their software, without solely depending on third parties to disclose vulnerabilities [11].

Due to funding provided by Microsoft in 2002, members of the PROTOS team created the company “Codenomicon” one year later. Codenomicon produces commercial fuzz testing suits. In the coming years several various fuzzing frameworks were created and published. Some of these are further described in Section 5.2.4, “Available tools/frameworks”.

File fuzzing became a very popular area of focus around 2004, mainly due to the discovery of a buffer overflow vulnerability in the JPEG processing engine, revealed by Microsoft[12]. Around year 2007 a major increase in the interest of fuzzing technique occurred among several different companies and even more fuzzing frameworks emerged, both commercial and freeware based solutions.

Fuzzing tools targeting specific areas of focus such as ActiveX and Web browser, were developed and released from around year 2000, some of which are illustrated in Figure 2: Brief History of Fuzzing [10] and discussed in section 5.2.4 Available tools/frameworks.

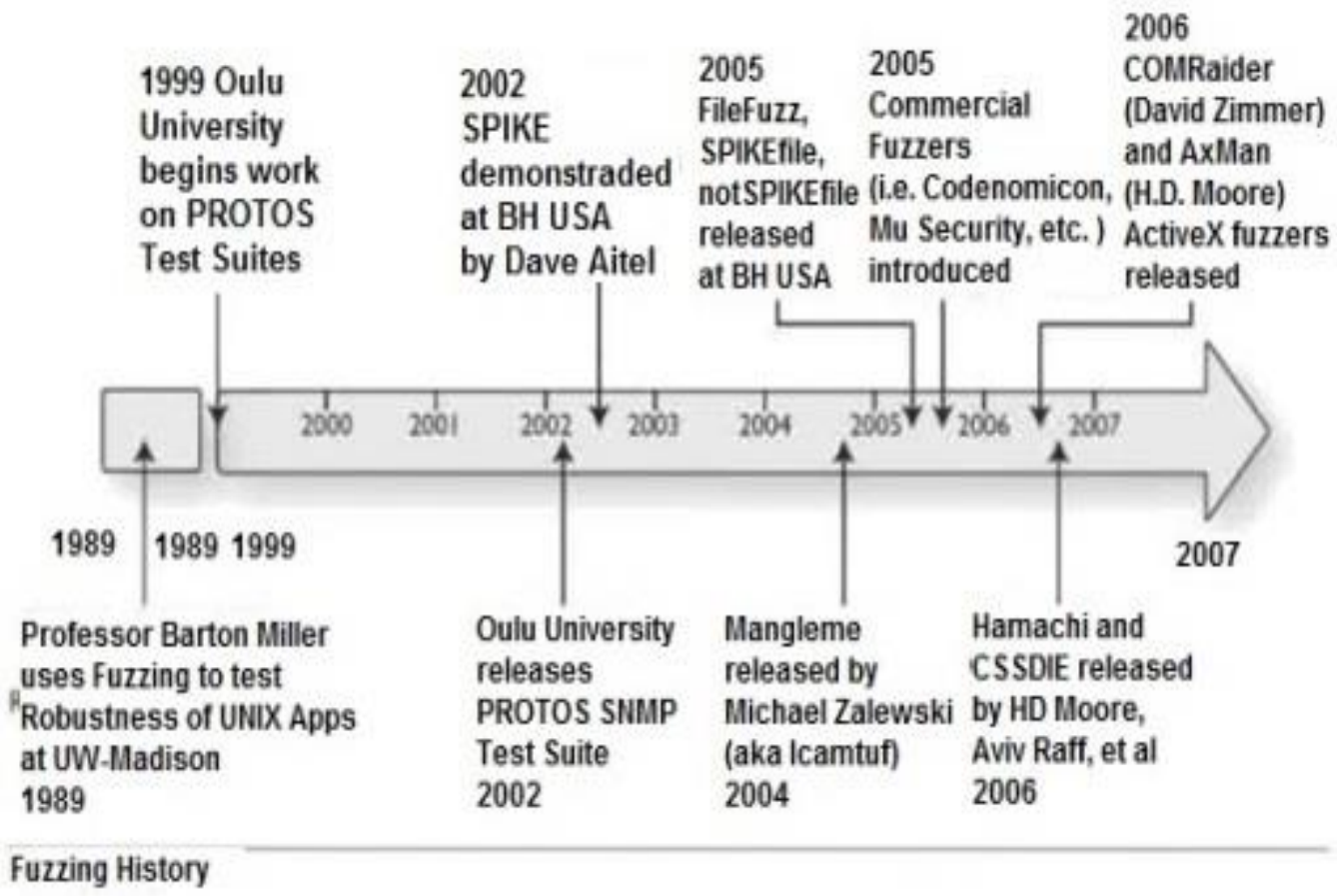


Figure 2: Brief History of Fuzzing [10]

As fuzzing is a relatively new automated testing technique, its history isn't very extensive. This also means that there is a high probability of new fuzzing techniques being developed in the future and that fuzzing will continue to evolve as well as being utilized to a greater extent.

As of today, fuzzing has become a rather well known technique and most companies are using or are in the process of deploying fuzzing in their development life cycle. It is said that as much as 80% (in 2009) of the leading service providers and device manufacturers are utilizing or planning to utilize some form of fuzzing[3]. Smaller software development companies are also slowly starting to make use of fuzzing, as it is becoming more and more well-known. Not too long ago (2008/2009), fuzzing was only associated with an unknown hacking technique that few quality assurance people knew about [3].

5.2.4 Available tools/frameworks

Due to the low level of human effort needed to discover new bugs in programs and the high number of discovered bugs with the use of fuzzing tools, fuzzing has become an important tool in the software security industry, and many major companies have included fuzzing while performing vulnerability discovery of their programs [1]. One of these is Microsoft with their SAGE white box fuzzer [7] which is deployed in the verification stage of their SDL[3] [13]. Another fuzzing tools which is used during software development, is the "jsfunfuzz" [14] tool used to fuzz JavaScripts, which has a central part in the quality assurance process of both Google an Mozilla[15]. Adobe also makes use of fuzzing in their "Secure Product Lifecycle", during the "Testing" step [16]. A model from Adobe showing their "Secure Product Lifecycle can be seen below:

[3] Security Development Lifecycle

The Adobe SPLC process

Figure 3: Adobe SPLC

Other fuzzing tools/frameworks have also been developed and made available for use.

SPIKE is an open source fuzzer licensed under the GNU General Public (GPL) license. SPIKE was developed by Dave Aitel as PROTOS evolved in 2002. SPIKE utilizes a block-based approach and is mainly intended for network applications and supports variable-length data blocks. SPIKE mainly operates by creating random data, but it also bundles various values likely to produce errors in commonly used network modules. As SPIKE was the first framework enabling users to effortlessly create their own fuzzers, the release of SPIKE marked an important step for the evolvement of fuzzing[10, 17].

Aitel also created another fuzzer named sharefuzz for UNIX systems. This fuzzer mainly targets the environmental variables of the system. sharefuzz incorporates a technique meant to simplify the process of fuzzing. The technique makes use of shared libraries to be able to hook functions returning environmental variables, swapping the original return value with long strings in the attempt of producing a buffer overflow.

Following the release of SPIKE, several fuzzers aiming for one specific target also saw the light of day. "mangleme"[4](2004) by Michal Zalewski, targeting web browsers by producing malformed HTML pages to be fed to the target Web browser, is one of these. The development of SPIKE was followed by another (Dynamic) HTML fuzzer called Hamachi, created by H.D Moor and Aviv Raff. The creators of Hamachi later teamed up with Matt Murphy and Thierry Zoller to create a cascading style sheet (CSS) parsing fuzzer called CSSDIE[10].

As stated, file fuzzing grew to become a major area of interest around 2004, and file format vulnerabilities also proved to be the best target for mutation-based fuzzing. This is due to the fact that there typically are a lot of samples available which easily can be manipulated and mutated while monitoring the target for crashes. Some popular file fuzzers are FileFuzz, SPIKEfile and notSPIKEfile[10, 18].

In addition to file fuzzing, ActiveX fuzzing also became a popular target, in 2006. At this time, David Zimmer released a fuzzer called COMRaider, followed by H.D Moore's AxMan tool. This tools target the ActiveX controls which could be accessed and instantiated by Web applications. Vulnerabilities in ActiveX software could be exploited remotely and represented a big risk, which also may be one of the causes for the big interest in the field[10].

There exist several other popular fuzzer tools and frameworks as well, such as Peach[19], but this paper will not contain a full list of all available tools.

5.2.5 Evolutionary fuzzers

Evolutionary fuzzing is a technique that makes use of feedback from program executions to improve further fuzzing runs. Code coverage, which gives a number of how much of the total code that has been executed, is one kind of feedback that can be used to improve the fuzzing process. Code coverage analysis is feasible and useful when application source code is not available. An evolutionary test tool receiving such statistics can use that information to actively learn the target's internal logic. Tools utilizing such processes can be referred to as grey-box fuzzing, and can be more effective at finding bugs than other fuzzing techniques such as black-box mutation fuzzing[20].

[4] Available for download at http://freecode.com/projects/mangleme [02/09-2013]

To be able to gather the needed information or statistics from the executed program, a fuzzer will typically have to be used in combinations with other tools such as debuggers and reverse engineering tools. To be able to benefit from the extracted information, several iterations will have to be performed where information from previous iterations are used to improve the next one, effectively making use of the fact that the process gets a broader understanding of the target program by each iteration as it is able to constantly learn more from the feedback.

One existing evolutionary fuzzer is the *Evolutionary Fuzzing System* (EFS), developed by DeMott, Enbody and Punch. This tool is built upon the General Purpose Fuzzer (GPF) and reverse engineering and debugging framework PaiMei. The EFS was tested against a communication protocol, and was able to learn the protocol language over time, while also discovering bugs in the protocol. EFS was able to learn the target protocol by evolving sessions: a sequence of input and output that makes up a conversation with the target. To keep track of how well the tool was doing it uses code coverage as a session metric[20].

The aim of the third research task in this project is to construct a fuzzer making use of code coverage and trace information for a given iteration to improve the strength of the samples used. This behavior is similar to that which identifies an evolutionary fuzzer. Even though similar solutions to the one which is to be presented in the third research task exists, I am not interested in creating an evolutionary smart-fuzzing tool solely to be able to make use of grey-box fuzzing, but also to be able to show results from the process and prove that the technique is working.

5.2.6 Limitations

Even though fuzzing might seem as the perfect tool for finding bugs and faults in programs, there are some limitations to the technique. The main limitations of using a fuzzer are that they will be unable to detect some types of vulnerabilities.

One vulnerability that might be missed by a fuzzer is a flaw called “Access Control Flaw”. When fuzzing a target program with some parts of the software only available to user with admin rights, the fuzzer might in some situations bypass the access control mechanism with corrupted data. However, the fuzzer has typically no way of knowing that it has entered a part of the software which should be restricted to users without privileged access rights, and will therefor continue fuzzing, without reporting the flaw. It is plausible to construct a fuzzer that are aware of such access control breaches, but such modification would most likely be very complex, require a lot of insight into the logic of the target program and the fuzzer would probably not be reusable with other software(without further modifications).

Even though most major companies (hopefully) do not enter many backdoors into their software and therefor do not have a big need of detecting backdoors during quality assurance, people trying to attack some piece of software might be interested in the

existence of a backdoor. However, backdoors is another flaw that fuzzer typically will miss. In situations where limited or no information about the target program is available, a backdoor will appear to a fuzzer like any other target logic. The fuzzer has no way to distinguish between “normal” behavior and that of a backdoor, as the program does not crash. For instance in a login screen, the fuzzer will have no way of identify a successful login attempt with a hardcoded password, when randomly using the hardcoded password, but will typically detect a flaw when supplying a malformed password making the login logic crash.

Some attacks on software consist of exploiting more than one vulnerability in succession to be able to achieve the desired action. For instance, one vulnerability may be exploited to get unprivileged access rights to a program, flowed by exploiting another vulnerability to escalate the privilege level further. A fuzzer might be able to detect the individual flaws, or only some of them, but in certain scenarios the individual flaws might not be considered to be a security risk. The fuzzer is unable to comprehend the concept of and identify such multistage vulnerabilities.

There are others instances of limitations of fuzzing as well, such as memory corruptions handled by a special memory corruption handler, but for the purpose of this paper, only some examples of limitations has be given. This is to illustrate some areas where fuzzers might not identify all flaws of a program and give some insight into fuzzing limitations.

[10]

As seen in section 5.2.2, some programs validate the input directly after receiving it from an external source and exits if anomalies are found. Such integrity mechanisms will abort execution when run through a fuzzer and is a serious limitation of fuzzing if no actions are taken to circumvent them.

When performing (file) fuzzing it is important to be aware of the fact that it in theory is very difficult to discover bugs. This is especially true with dumb black-box fuzzing. This is due to the fact of the enormous number of different mutations available. When few bugs exist, and they might only be triggered by one single input combination, a large number of fuzzing iterations will have to be done to discover it. The number of possible mutation for a given file may seem infinite, but there is indeed an upper bound for all files. Take for instance a file of 2 KB. To be able to do all possible mutations, there have to be performed 2 ^(2048 * 8) fuzzing iterations to exhaust all possible combinations of the bits in the file. As the files get bigger, the total number of mutations increases to a number of needed fuzzing iterations which is impractical to perform. It is due to this fact that the need for smart decisions and key areas to focus fuzzing runs is highly important. This aspect thus impacts the efficiency of a given fuzzer. To be able to use black-box fuzzing with no knowledge of the target program while still having a decent bug discovery probability, there is a need to do smart things during the fuzzing process.

Finally one also have to keep in mind that if the target program of a fuzzing process has been fuzzed earlier, during the testing phase when the program was developed, this might heavily impact the results when attempting to uncover faults with a fuzzing tool. As different fuzzing tools typically discover some of the same types of faults, some of these will be likely to be fixed before the program was released, greatly reducing the effectiveness of a new fuzzing run in regards to the number of unique crashes discovered.

5.3 Instrumentation

"The term instrumentation refers to an ability to monitor or measure the level of a product's performance and to diagnose errors." – Microsoft [21]

"Instrumentation for a software application is similar to the gauges and instruments on the dashboard of a car—it provides the information you need about the internal functioning of your application." – OC System[22]

5.3.1 Concepts

The concept of instrumentation is to be able to take control over any given program, in the sense that one is able to alter the execution of the program. Instrumentation provides an ability to insert, or remove, instructions into any given program. The practicalities of how and where new instructions are inserted (or removed/changed) depends somewhat on the implementation of the instrumentation tool, but common targets, or hooks[5], are for instance at the entry- or exit points of a function, basic block[6][23] or section. One can also perform instrumentation with each instruction as the target. Instrumentation can be used, as mentioned to alter or remove instructions, which certainly will change the target programs behavior. However, this is rather hard to achieve without corrupting the program or its data, especially in a generic fashion. Due to this, instrumentation is typically used to monitor and log program behavior during execution.

There are mainly two different types of instrumentation techniques. One technique is to add the instrumented code dynamically while the executable is running. This technique is also known as Just in Time instrumentation, as the instrumented code is inserted right before it is executed. The main strength of the dynamically added code technique is that no source code is required for instrumentation. In some cases, depending on the implementation of the instrumentation tool, the tool might also be able to attach itself to a running application. The other technique which is somewhat less common is to add the code statically before execution.

[5] Here: A place in the program where one could perform instrumentation, for instance by inserting instruction(s) at the location in question.
[6] A block of code with a single entrance and a single exit point

The scope for instrumentation is rather wide as one basically has the possibility to insert an arbitrary program into the target program. One usage of instrumentation is to uncover the execution trace of a program or to calculate the code coverage of a program. Other interesting information can also be extracted, such as the total number of instructions in the program or how many times an interesting function is entered. Even the content of the registers may be extracted dynamically during execution.

Typically there is some extra overhead when instrumenting an application which may increase the required run time of the target application. In addition, depending on how much new code is inserted and where it is inserted, the instrumented code can also greatly impact the total running time of the target.

5.3.2 Tools

There are several various instrumentation tools available today. One of the most common one is Intel's PIN[24]. PIN needs a "PIN-tool" file written in C++ or C to know what and where to instrument, and what to do at the instrumentation locations and data produced. PIN is a so called JIT (Just-In-Time) instrumentation tool, meaning that the additional code will be inserted at the appropriate locations in the program during execution, just before reaching the instruction in question.

Some compilers even have the option to activate instrumentation during compilation of source files. The Linux C compiler "gcc" (GNU compiler toolchain)[25] is one such tool which provides means to automatically instrument the desired parts of an application[26].

Another fuzzing tool which is based on dynamically adding code during execution is "Aprobe", as expressed by the developers: "It provides a way to transparently insert new code at runtime, while the application is in memory"[22]. Aprobe needs a "patch" or a "probe" where the user has specified the changes needed or the data to be collected. These probes are expressed in C or Java.

In theory PIN and Aprobe seems to have much of the same functionality and strengths. Among the most important features these two instrumentation tools support are:

- No changes are made to any application files
- No source code is needed
- Entire application can be traced
- Users may define their own instrumentation

Also C is supported as a language to express or define the specific instrumentation to be done. Each tool also supports one additional language (even though C++ might be seen as an extension of C), but these differ; Java and C++. I have not had the possibility to test these

two tools against each other as that falls outside the scope of this thesis and I only need one tool to perform my research.

Xept is yet another instrumentation tool and language. This tool is somewhat too restricted in its area of operation for my needs. It focuses on identifying and fixing bugs in programs where source code is not available. Xept can modify object code so that it may detect, mask, recover and propagate exceptions from for instance shared libraries.

Of course the list of instrumentation tools does not end with the few tools discussed in this subsection. Many more exist, and some may have strengths which others lack, which one is the best depends on the needs and requirements of the situation.

5.4 Minimal set

In certain situations one have a set of templates or samples which are to be used in calculations or executions of some kind, normally in conjunction with a target program that accepts such samples. As the various samples contain different information, it is typically useful to have many samples to be able to get a broader coverage of the operations they are to be used for. As these sets of samples can get rather big, the time needed to use all samples in the processes in question can get quite long. If it could be possible to reduce the initial set of sample to a smaller one without losing any total coverage, this would have a positive impact in the total time needed when feeding the samples to the target program. Such a reduced set of samples are called a minimal set, or minimum set, and is desirable to reduce both the size of the set, but most importantly the required time when working with the set.

It is possible to reduce the original set of samples due to the fact that some samples contain data that trigger the same parts of the program, or results in the same coverage. In such cases only one of these samples are needed and the other ones can be excluded from the minimum set. To be able to create the most optimal minimum set, the coverage information for each sample needs to be saved, making it possible to compare the information and pick the different samples needed to produce the maximum total coverage with as few samples as possible. To be able to retrieve the coverage for each sample, instrumentation is typically used to be able to track the code paths taken by the current sample.

One example of a tool that extracts a minimum set is a tool bundled with the Peach fuzzing framework. The tool requires a folder of the original set as well as a path to the target program which the samples should be run through and tested against. The result of the operation is a new folder containing a subset of the original samples which yields the same total coverage of the target application as the original. The samples in the produced samples are the minimum set of the original set for the provided target program.

6 Presentation

6.1 Target programs

The choice of target programs which I will perform fuzzing and research on is important. The targets I chose will not be the newest and up to date versions, as there will be more bugs in earlier releases, which in turn lead to more crashes being detected. For some of the objectives, such event should prove interesting data material for analyses. I will use two different target programs. The primary reason is to be able to extract a much greater dataset, which in turn is favorable to have a more solid base to draw conclusions from. Another reason is to be able to discover drastic differences between the code coverage and unique crashes with a given template used with the first program and the second. The target programs are both freeware, so they should be publicly accessible if one would like to test these programs. Each target program should be simple enough to be suitable for fuzzing and complex enough to produce a large set of different crashes within the time frame of this thesis.

6.1.1 SumatraPDF

The first target program I have chosen, is a program with a graphical interface, **SumatraPDF** [27]. As the name indicates, Sumatra PDF is a PDF reader, a lightweight PDF reader. I have chosen this program as my first target as I wanted at least one target with a GUI[7] as such programs utilize additional code for the graphical part which I wanted to include in the research. Also, as PDFs are in no short supply, I was certain that finding samples would not be an issue for this program. As applications with a GUI normally takes somewhat longer to both instrument and fuzz, than command line programs, the GUI target program of my choice needed to be as lightweight as possible, which Sumatra suffices to a great extent. I have chosen to utilize version 1.5 of SumatraPDF even though version 2.4 is already out, to make sure to get a sufficient number of crashes during fuzzing. The oldest available binary of SumatraPDF is version 0.2, and the reason why I did not choose this version is that I wanted the earliest “complete” version of the program which typically is indicated by version 1.0. It would therefore be desirable to utilize version 1.0 instead of 1.5, but due to the fact that the earliest versions of the SumatraPDF executable seemed to be packed with UPX[28], some problems arose when trying to disassemble the binary, which in turn lead to not being able to perform some of the critical static analysis tasks required. Sumatra PDF solely consists of the SumatraPDF.exe file and no other files such as libraries are needed to run it. SumatraPDF was introduced to me and suggested as a target program by my external supervisor to this thesis.

[7] Graphical User Interface

6.1.2 xPDF

The key is to get a dataset which is as broad as possible, that cover as much code and functionality as possible and which is created with as much variations as possible. To achieve such a dataset, I searched for target programs that were as different as possible in as many aspects as possible. Therefore, my second target program had to be a command line program. To simplify certain aspects of the research tasks of the thesis, the second target program had to be one which could accept the same kind of input files as the first target program, PDF files. Desirably the program should also do some sort of file parsing as parsing corrupted files might cause crashes. A command-line file processor is also a more suitable target for fuzzing as it takes a file as input, does something based on the content of the file and then exits, which typically is faster for a console program than a program with a graphical user interface[29].The program I have chosen as my second and final target program is **xpdf[30]**. This is a program which has several features, but I'm only going to investigate one of these, which parse PDF files and extract their content for saving in a plain text format. Conveniently this feature has been packed together in its own executable, making it easy to separate uninteresting code from the interesting as well as it will improve various analyses (i.e. the main executable does not rely on any shared libraries, other than the Windows specific ones). The program is also rather efficient and is therefore suitable for my purpose. Xpdf is licensed under the GNU General Public License (GPL), version 2. The currently latest version of the program is version 3.03, but I will be using version 2.03 (version 2.03 downloaded from[31]) as more bugs should exist in earlier versions. xpdf was found during an open search for command line programs parsing PDF files. The most common command of this target program that will be used is:

```
xpdf.exe [options] <PDF file> [<location to save the output>]
```

Figure 4: Second target program's command with arguments

Argument	Usage/meaning
Options	Various options to specify the behavior of the program
PDF file	The path to the PDF file to convert
Output location	This is optional, and specifies where to save the results of the conversion process.

Table 2: Second target program's argument description

6.2 Metrics

During the various research tasks conducted during this project, different metrics have been used to measure the processes that are run and the executions that are performed. For some cases there may be a specific metric used, which will be explained in the given section. However, on a more overall level there are mainly three major metrics that are utilized; Code coverage, time and unique crashes.

6.2.1 Code Coverage

Code coverage is one of the main metrics when measuring the traces of an executed program. The concept of code coverage describes how much of the total code that has been reached or touched during an execution and is often expressed as a percentage of the total code. Code coverage is used to a great extent when attempting to evaluate if the amount of code of the given application that has been executed when run with various input files. Code coverage is generally calculated by having the number of execution units[8] which was indeed executed during a run, divided by the total number of execution units.

While code coverage generally gives a good indication of the total code that has been executed, it may also be somewhat misleading in certain scenarios. When extracting the code coverage of an execution, one typically does so to investigate how effective the execution was with regards to reaching as much of the code of the program as possible, while also collecting the code coverage of several successive executions to see which covers the most of the programs code. In this scenario it is important to note that two executions with identical code coverage may cover two completely different, but equally large, parts of the actual code. To further illustrate this scenario consider the following example.

Execution units executed in execution 1: 7ac1, 7ac3, 7ac5, 7ac7. Code coverage: 8 %

Execution units executed in execution 2: 7ac1, 7ac3, 7ac5, 7ac7, 7ac9. Code coverage: 10 %

Execution units executed in execution 2: 7ac1, 7ba1, 7ba3, 7ba7. Code coverage: 8 %

Execution 2 looks to be better than execution 1 as it has higher code coverage, which is true. When considering execution 3 against execution 1 and 2, it seems as it is equally good as execution 1, but worse than execution 2. This may however not be completely true, as execution 3 has executed very different parts of the program than the first two executions. In such scenarios it could be valuable to consider both execution 2 and 3 as almost equally good, as one typically aims at reaching as much of the code base as possible. Furthermore if only execution 1 and 3 were recorded, one might discard one of them if only looking at the code coverage as they seem equal, but this would lead to loosing total coverage. If the traces used to produce the various code coverage numbers are available, set theory could be utilized to assist in giving a more realistic and rewarding view.

[8] Execution unit depends on the desired level of granularity. Examples are functions, basic blocks or instructions.

This shows that code coverage alone might not be a sufficient metric when investigating code executed, and should be complemented by more sophisticated analysis when aiming at getting the best coverage results possible. Even though code coverage may suffer from this drawback, it can still serve as a good indication of the code executed.

To calculate the code coverage while performing the research required by the various objectives in this thesis, I will utilize a python script which reads two text files with information about the executed units(retrieved through instrumentation) and information about the total number of execution units for the program (retrieved by static analysis of the program).

6.2.2 Time

For some of the research to be conducted it will be interesting to look at the time needed to perform some operations. This gives a good indication of the complexity of the process as well as how much required time it is reasonable to anticipate for the given operations. Elapsed time can also be used to compare different processes.

Typical ways of measuring time is to look at the time used by the processor for a given process, but also the actual elapsed time.

6.2.3 Unique Crashes

As fuzzing was used to a great extent in several parts of this project, and fuzzing is all about producing as many crashes as possible, unique crashes will be an important metric. During a fuzzing process, a given crash can be produced several times from different inputs. As the aim of fuzzing is to discover faults or vulnerabilities, it is not very interesting to see how many times a given fault can be discovered[9], but rather how many different (unique) crashes that can be discovered. Due to this fact, the number of unique crashes will be used as a metric.

6.3 Tools to be used

6.3.1 Fuzzing

As the fuzzing tool of my choice for the practical work of this thesis I have chosen "Peach"[19], developed by Michael Eddington of Déjà vu Security [32]. Peach is not actually a fuzzer itself, but a fuzzing platform/framework providing an XML and Python hybrid way of quickly creating a fuzzer that may be used in various scenarios and on different data formats. Peach is a "Smart Fuzzer"[10] that is capable of performing
both generation and mutation based fuzzing, which is one of its great features. Peach requires the creation of Peach Pit files that define the structure, type information, and

[9] It may be somewhat interesting to see how many times one specific crash has been discovered as many occurences typically indicate a fault that is easy to trigger and should be focus on fixing.
[10] A fuzzer with more intelligence than a standard dumb fuzzer, which is generating and testing completely random data with no insight in the targets internal logic or the expected data format.

relationships in the data to be fuzzed – how the valid data look like. Peach can then use this definition often along with a sample file, to generate many interesting variants of invalid data. Peach is moderately complex and somewhat poorly documented, however the existing documentation on the Peach site is very useful. In combination with various examples and some work, the pros of peach heavily outweigh the complexity issue.

Version 1 of Peach was developed in 2004 at ph-neutral 0x7d4. The first version of Peach was a Python framework for creating fuzzers. In 2004, during the summer, the second version of Peach was released. This version was the first comprehensive open source fuzzer that included process monitoring and creation of fuzzers using XML. The last version of Peach, version 3, was released early in 2013. This release differentiates to a great extent from the earlier versions. First of all, this version was a complete rewrite of Peach. The major change is moving away from Python and instead using the Microsoft .NET Framework, primarily C#. The version used was version 2.3.9, as this version consists of desirable feature while also accepting my current Pit Files as is. For the last research task I utilized Peach 3.0 as the speed has been greatly increased and the work to be done in the last research task will be very time consuming. Due to some changes that have to be made, the working Pit files for version 2.3 had to be changed to be accepted by the 3.0 version of Peach.

Peach has, among many other features, the possibility of using various debuggers to increase its capability of capturing information related to a crash. When running peach to perform fuzzing in the various research cases, I made use of a debugging extension called “!exploitable”, which is further described in section 6.3.4:!exploitable

6.3.2 Instrumentation

The practical work related to the research of this thesis required the use of instrumentation to be able to monitor and log the activities and behavior of executed programs. The tool used for this purpose was Intel’s PIN instrumentation tool[24]. PIN was chosen both for its extensive API[11] and its possibility to perform analysis at many different granularities. PIN is also a tool for dynamic instrumentation which best suits m needs. Another positive aspect of Pin is that it is made to be portable. It supports Linux binary executables for Intel Xscale (R), IA-32, Intel64 (64 bit x86), and Itanium processors; Windows executables for IA-32 and Intel64; and MacOS executables for IA-32[33].

PIN is a JIT compiler, recompiling sections of the original code with additional/new instructions. These instructions come from a Pin tool file. PIN’s inserted code can be in the form of analysis routines or callback routines. Analysis routines are called when the code they are associated with is run. Callback routines are called when specific conditions are met, or when a certain event has occurred.

[11] Application Programming Interface

On Windows systems, Pin requires Microsoft Visual Studio[34], or similar software to be installed to support compiling of C or C++ code as well as to run the nmake files bundled with the PIN framework.

6.3.3 BinNavi and IDA

BinNavi is a binary code reverse engineering tool which can be used for some graphical representation or other analysis tasks[35]. It is mainly written in Java. BinNavi also supports a scripting API making it possible to create specialized scripts to use when working on the binary data. The actual disassembly process is delegate to another third party program; IDA Pro[36]. The disassembled database produced by IDA is imported into BinNavi and stored in a database structure. Depending on the version of BinNavi used, the database is either a PostgreSQL database or a MySQL database. The version of BinNavi that I used was utilizing the PostgreSQL database. The scripting option is one of the major reasons why I chose BinNavi as one of the tools to work with. Even though IDA Pro also supports a scripting API and mostly contains the same information, as its exported data are imported into BinNavi, I chose to use BinNavi as its scripting API seemed somewhat easier to use and will fit my needs better.

IDA Pro is unarguably a complex program packed with various features. The full description of IDA is "IDA Pro combines an interactive, programmable, multi-processor disassembler coupled to a local and remote debugger and augmented by a complete plugin programming environment"[37]. The disassembly part of the tool is the feature I used, to extract information about the target program's internal logic. The goal of a disassembly process is to reverse the steps that were taken when the target program was created. The tool used to undo the assembly process of creating an executable program is called a disassembler, and the outputs or results of such a disassembly operation is a set of assembly instructions. If also a decompiler is used, one attempts to reproduce high-level programing languages based on assembly language as input. However decompilation might be rather difficult due to various reasons[38]. The information uncovered by IDA pro can be saved into an IDA database file with extension .idb for further work with the target program without having to redo the previously discussed operations, but it also enables the possibility to export the information to other application able to interpret the .idb file format.

6.3.4 !exploitable

As previously mentioned, "!exploitable" is not a standalone tool but an extension to the Windows debugger (Windbg[39]), used by peach when logging information when crashes occur. !exploitable is pronounced "bang exploitable" and provides automated crash analysis and security risk assessment. The tool has the possibility to distinguish between unique crashes as well as to automatically assign an exploitability rating for the various crashes. The ratings used are: Exploitable, Probably Exploitable, Probably Not Exploitable, and Unknown. This debugging extension can be used to extract information about the current state when a

crash occurs, such as stack traces, heap data, register values and more. !exploitable was developed by Microsoft[40].

6.3.5 Immunity Debugger

Even though the fuzzer Peach is able to make use of Windbg and its extensions, it is also useful to have a standalone debugger tool to investigate and validate various aspects and results. I have chosen to use Immunity debugger as my standalone debugger tool, as this is a powerful debugger with a great user interface, but also due to its ability to run custom made python scripts to easily extract various information. This feature will be utilized to some extent during static analysis.

6.3.6 010 Editor

010 Editor is a tool combining the features of several standalone editors. It can function as a normal text (and high level code) editor, but it also has the possibility to edit hex files. In addition some disk and process editor features are included. When it comes to hex editing, the 010 Editor stands out as opposed to traditional hex editors which only display the raw hex bytes of a file, while the 010 Editor can also parse a file into a hierarchical structure using a binary template. The results of running a binary template are much easier to understand and edit than using just the raw hex bytes. Another powerful feature of this tool is the possibility to load two files and compare them automatically. Differences, matches and segments that only exist in either of the files, are automatically identified. Furthermore, the editor is also scriptable, meaning that you have the possibility to create your own scripts which can be run through the editor, tailoring the editor to do exactly what is needed. Several more features and options exist. The 010 Editor is a commercial program, but a 30-days evaluation option is available[41].

6.4 Working environment

The majority of the practical work was conducted on a Virtual Machine running XP. The major reason for performing the work in a Virtual Machine is that with XP no ASLR[12] is present, which certainly would complicate the process of comparing addresses of a given execution with another execution of the same program. There are also some third party dependencies required for the fuzzer and instrumentation tool which are preinstalled in the virtual machine, such as Visual Studio. Running in a Virtual Machine also has the benefit of snapshots[13], as well as protection of the host from potential errors or problems.

The specifications of the Virtual Windows XP machine are as follows:

[12] Address Space layout randomization, a computer security technique which involves randomly positioning key data areas of an executing program, such as the base address, the stack, the heap and any potential libraries.
[13] Restore points to which it is possible to revert the state of the virtual machine

Field	Value
Operating System	Microsoft Windows XP Professional
CPUs	1 CPU – 2 cores from host
RAM	3 GB from host
Storage	30 GB from host + shared folder on host drive

Table 3: VM specifications

As the speed and performance of the virtual machine greatly relies on the host machine, its specifications follow:

Field	Value
Operating System	Microsoft Windows 7 Ultimate
CPUs	Intel Quad Core i7-3820 @ 3,7 Ghz Socket-LGA2011
RAM	16 GB DDR3 2133 MHz
Storage	Total storage: 7,5 TB. Drive used by VM: 1TB SATA-600

Table 4: Host specifications

The Virtual Machine also has a shared folder with the host, where files such as logs and samples will reside. This shared folder is mapped as a network drive (Z:) on the Virtual Machine.

6.5 Tool interaction- and program flow overview

This section is meant to briefly describe how the different tools are used and connected during some of the work to be done in a way that is easy to perceive. It will contain some flow charts and figures describing the sequence of operations for various techniques and work cases.

6.5.1 Peach operation flow

Firstly an overview of the internal operations of a normal standalone fuzzing sequence is presented:

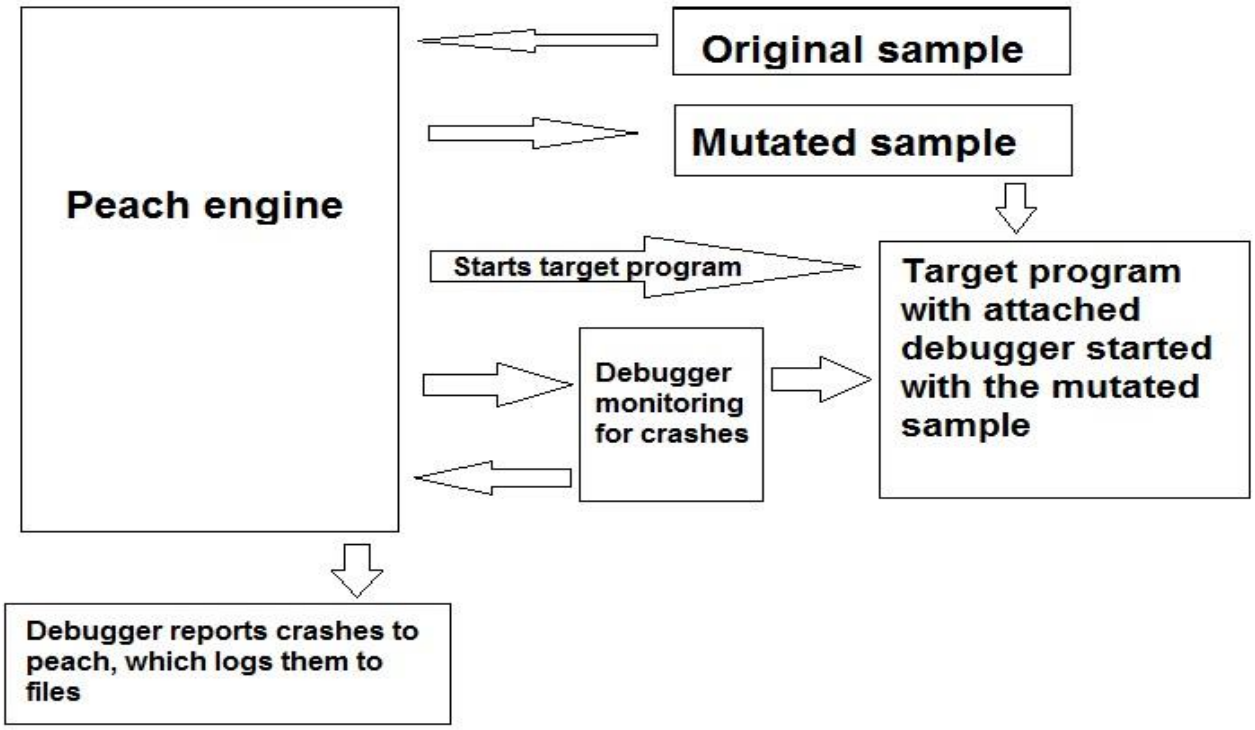


Figure 5: Internal peach fuzzing operation

The presented figure shows the operation of one iteration in peach during normal fuzzing. The technique used to create the mutated sample changes for each iteration. Also the target program is automatically closed by peach when it has finished its work.

6.5.2 Getting trace and code coverage information operation flow

The next figure to be presented contains an overview of how to produce trace and code coverage information for a given program execution, with a mutated sample.

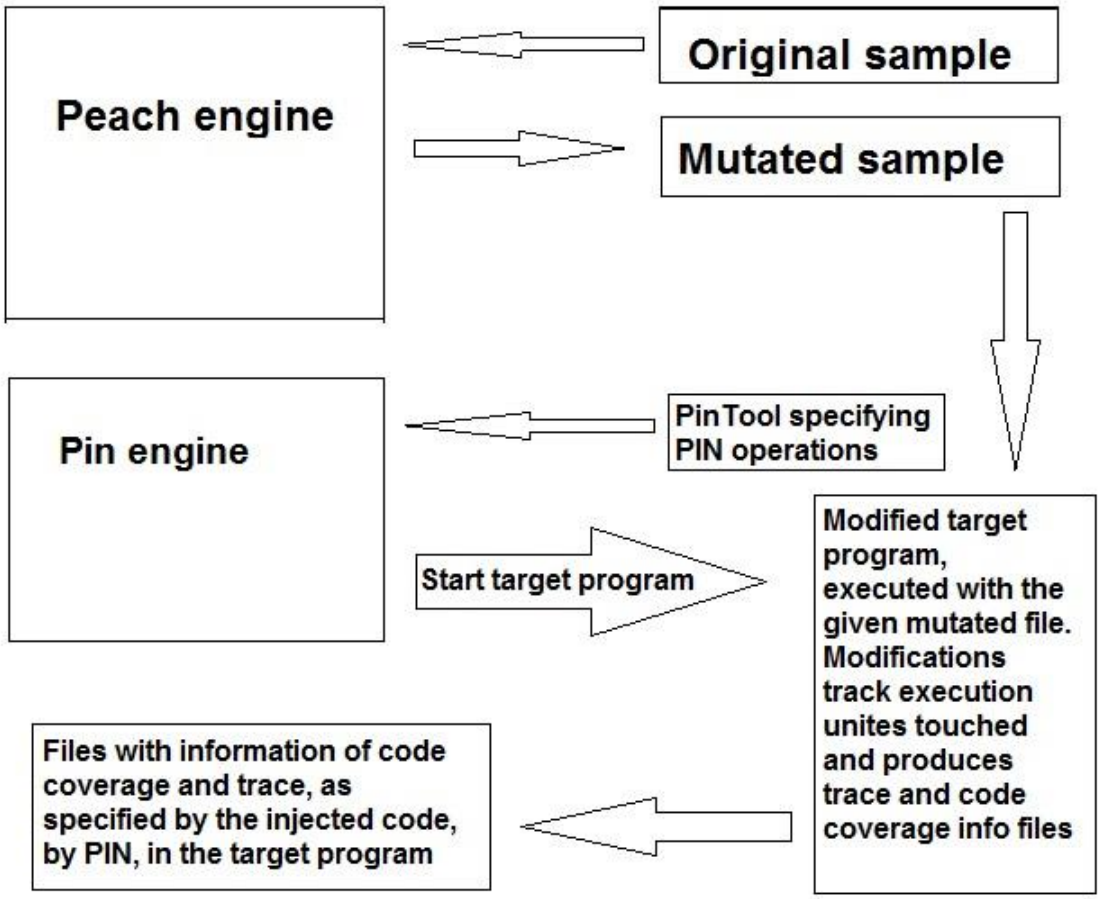


Figure 6: Getting trace and CC information with mutated samples

Pin and Peach can and will have to be started separately, but the results from a mutation process in peach can be used when running PIN by providing the produced mutated sample from peach, when starting the target application through PIN. The Pin Tool required when starting an instance with PIN specifies what the PIN engine should do to the target program during execution. In this instance firstly all execution unites are counted, before execution begins. When execution starts, all execution unites that are indeed executed when running the program are recorded. At the end of the execution, code coverage is calculated based on the saved values and it is printed to a file together with the addresses of all execution units recorded.

The two presented figures contain a very brief overview of the operations. Additional operations may however occur, but should be specifically described when applicable.

6.6 Assumptions and decisions

6.6.1 Thesis division

As this thesis investigates different aspects at various stages of the fuzzing process, it seemed natural to split up the actual work to be done in smaller research cases. This division makes the layout of the thesis clearer and it is easier to quickly find the relevant parts of the paper for different scenarios. The division was made based on the stages of a fuzzing process, but also on the functionality discussed in the various cases. Due to the amount of result data, it is also clearer to view the results section and find the appurtenant research case according to this division.

6.6.2 ASLR

As stated, I chose to work on Windows XP to circumvent the ASLR of newer operating systems. It is however possible to turn off ASLR in newer systems, such as Windows 7(by using the Enhanced Mitigation Experience Toolkit) [42]. Even though it is possible to deactivate ASLR in newer systems, I decided to use an operating system where there was no such support at all, to be completely certain not to get any interference. Also most XP version utilizes 32-bit technology as opposed to newer systems where 64-bits are more common, which is desirable to further minimize potential sources of errors.

6.6.3 !exploitable and it's categorizing of faults

When the fuzzing process uncovers a fault, the debugger extension !exploitable categorizes the faults as; Exploitable, Probably Exploitable, Probably Not Exploitable and Unknown. When analyzing or doing further work on the fuzzing results, I utilized all unique faults detected, regardless of their category, including Probably Not Exploitable and Unknown. The reason for this decision is that typically I was not interested in actually fixing or detecting actual vulnerabilities, but rather to research and uncover typical behavior and patterns regarding the crash locations, the influence of various sample characteristics and similar properties of the fuzzing process, which can be done regardless of the fault actually being a vulnerability.

6.6.4 Iteration count

When performing fuzzing on either of the target programs, I typically ended the process after roughly 20 000 tests were done. This amount, in my opinion, is not really sufficient to cover all the various features of the target program, and also typically will not cover all available samples depending on the switchCount and the total number of samples. However to be able to performed the needed operations within an achievable and acceptable time frame I typically decided to use the mentioned number of iterations. The reason why I at all had to end the process is that the fuzzing process is in fact never-ending.

6.6.5 Unique crashes

Occasionally one vulnerability might be discovered several times, for instance with different input samples taking the same path with the same values. This leads to two different faults

being detected and recorded, even though they are valid for the same vulnerability, and thus also the same crash location. Typically only unique crashes were interesting for the thesis, but depending on the current research goal, this might differ. Luckily, the unique crashes are easily accessible as they are organized in a fashion where each unique crash is easily distinguishable from others, and also each unique crash contains information about each test case that uncovered the crash. The reason why I typically was not interested in each fault that caused a given unique crash is that most of the information regarding that crash is similar for the various faults that caused it, for instance the crash location, and therefor only examining one fault for each unique crash is typically sufficient.

Section II

7 The work process, results and analysis

This section of the paper primarily contains a "report" of the work that has been done during the practical aspects of the Master's project, including specific commands used, steps taken, procedures and tools utilized. New key aspects and technologies encountered are also explained. The work conducted in connection to the various research tasks are described in detail, in the order that each research task was presented in Section 4.3.1: "Nature of the thesis". The work process is presented in detail to give the possibility of recreating the experiments if necessary. The presented material may among the description of the steps taken and techniques used, include some code or command snippets from the research process whenever appropriate.

While investigating the various research tasks of this thesis, sample files were needed. As both target programs are able to handle PDF files, naturally the samples consisted of a collection of PDF files. To be able to cover as much of the functionality of a program parsing the PDF files as possible, many different PDF files with different content were needed. I was able to acquire a large set of PDF files with very varying content, which I used as my sample files, for both programs. In total, 6064 different PDFs were acquired, and were stored in the shared folder between the VM[14] and the host. The reason why I used such a great number of samples, is mainly to increase the chances of achieving a rather high level of code coverage (and thus hopefully uncover more bugs), when the program is executed with all samples.

During the practical work, the work process for most of the research tasks involved sample files used in conjunction with the two target programs. Typically a large set of sample files are desirable, but by having such a large set of samples, the time required to do each operation will in most cases increase as the operation will have to be performed for each sample file. It would therefore be desirable to minimize a large set to a smaller size, but without losing the benefit of having a large set; greater code coverage. This technique is often referred to as a minimum set. To create a minimum set, one executes the target program with each sample file and records the various pieces of code of the target program that gets touched with that specific sample. At the end, sample files which solely have touched parts covered by other sample files, are excluded from the minimum set. In this way, it is possible to exclude samples whose code coverage is already covered by one or several other samples. The result is a set with the minimum required samples still giving the maximum code coverage. For my thesis, I performed a lot of time consuming work including fuzzing with all the samples. This is true for several of the research tasks. To take the time to construct a minimum set and potentially get a smaller working set of samples would be ideal and beneficial for later steps in the work process.

[14] Available from VM at "Z:\Host storage\samples\pdf\"

In the following subsections, “original trace” is used at certain occasions. An original trace (for a sample) is simply the first trace of the execution of a target program with one specific (non-mutated) sample. When having performed fuzzing of a target with various samples, an original trace is typically used in combination with the trace of a changed (mutated) file, when for instance comparing the two traces for differences or other interesting aspects. The trace for a program with a sample before mutation is referred to as an original trace, while the trace for the same program with a mutated version of the same sample is no longer an original trace.

When performing program analysis, such as monitoring, code coverage and fuzzing, one typically has to choose a level of granularity for the investigation. When deciding upon what to use when doing the research of this thesis, I had to decide between basic blocks, functions or instructions. I found out that the choice had to be based on what granularity was used by third party tools needed, as well as on what I was trying to achieve or discover. I also recognized that those needs might differ from one research task to the next, and therefore decided to make the decision separately for each of the major research cases of this thesis. The exception to this rule is when instrumenting a program with PIN. PIN has built in methods to identify among other components functions and the main executable. In my experience however, these methods are not always as reliable as one would desire, both because PIN is not always able extract sufficient information from the binary to make the correct decisions, but also because PIN treats certain code components differently from the traditional way, specifically, but not necessarily limited to, basic blocks. It does however seem that PIN is always able to fully detect and instrument all instructions of a program, and due to this fact, I primarily chose to work at the finest granularity level when using PIN. This means that if I want to identify function start addresses or other aspects of the code, I have to go through the recorded instruction data from the instrumentation and make use of static analysis of the binary to match certain instruction addresses against static analysis results to retrieve more information of the instrumented results. There is one important trade-off to be aware of when working at the finest granularity level, even though the output is right and everything is collected, the instrumentation process might take a very long time, greatly depending in the complexity of the analysis function which is called for each executed instruction of the original program. Also if there is a need to store and/or inspect a property of the instructions, it is important to choose an effective data structure to hold the data. For research cases where the required processes are expected to be very tedious another granularity level may have to be chosen.

The term research task will be used in this section, and refers to one of the main focus areas of the research for this project. As described in Section 4.3.1, Nature of the thesis this project is divided into multiple research tasks and each individual research task is described in Table 1: Research tasks.

For several of the research tasks a minimum set of the samples to be used is needed. Before presenting the work process for each individual research task, I will present the process performed to extract the minimum set. As briefly mentioned this is, among other reasons, useful as one ends up having a set which is smaller while still covering as much code as the full set of sample files. To make the minimum set of PDF samples to use for the SumatraPDF target program, I utilized a tool bundled with the used version of the peach fuzzer called minset.py. This tool discovers the code coverage of a target program for each sample file available, and then creates a minimal set of the sample files, which in combination yields the greatest code coverage of the program. The full command used can be seen below. After several days of running minset, the operation finally completed and the resulting minimum set had drastically reduced the number of samples from 6064 to 842. The time needed before discovering crashes when fuzzing with this new set should accordingly have been reduced drastically compared to the full set of samples.

```
python minset.py –k –s “Z:\Host storage\samples\pdf\*.pdf” –m minset
E:\Dropbox\Master\Other\SumatraPDF.exe %s
```

Figure 7: minset command

Argument	Usage/meaning
-k	Kill the process when CPU time is near 0
-s	Path to sample(s)
-m	Path to store the minimum set of samples
command.exe	The program to run the samples with/within
%s	Substituted by the filename for the current iteration

Table 5: minset command’s arguments

The minset operation when run with the second target program reduced the number of samples from 4900 to 245. The reason why the initial number is lower for this target program, than for the first target program, is that some files were rejected by the program before starting to monitor those files and some files had to be removed manually as they made the target application crash.

7.1 Code Coverage

- *How far from a trace does a bug actually occur?*

This section deals with the first research task; code coverage.

7.1.1 Work process

The reason why I focus on this question in this thesis is that if a bug, or rather the majority of bugs, normally occur close to the original trace of a given program, it means that having a high percentage of code coverage while performing fuzzing will uncover many more out of the total vulnerabilities and bugs in the programs. If most bugs are located close to the original trace, and we have an original trace with high code coverage, there is a greater chance of uncovering more bugs. However, it is important to keep in mind that for some bugs that are close to the original trace, might still be hard to reach due to complex program flow mechanism which could be hard to trigger when fuzzing. With this point in mind, and if bugs are found to typically be close to the original trace, more bugs should be easier to touch when having a good code coverage.

While researching this research task of the thesis, I decided to investigate the execution at function granularity as this gives a more detailed view on the program, and is easier to use when calculating the number of hops between where a crash occurs and the original trace as the traces will be generated based on functions touched.

To investigate this research task, I performed two steps that had to be done only once, followed by two other steps that had to be repeated for each crash detected. These steps are:

1. Steps performed once:
    a. Extract all functions of the target program and their call targets
    b. Get the end address of the image/module in question
    c. Fuzz for crashes with the files from the minimum set, while recording the original file from which the mutated file was created in which the crash occurred as well as the function in which the location of the crash resides.

2. Steps repeated for each crash:
    a. Rerun the target application through PIN with the original file from which the mutated file causing the crash was created. The PIN tool used will extract all functions touched during execution.
    b. Calculate the number of hops between original trace for the original file and the crash in question.

Step 1a is be performed through a script given to BinNavi preloaded with the disassembled info for the target application to get a function call graph (non-visual) for the entire program. Step 1b is achieved with the use of Immunity debugger and a custom script[15]. Step 1c is done with Peach fuzzer and a peach pit file pdf_semi_smart_sumatra.xml [16] to find and detect crashes and get information about their location. Step 2a is done through PIN using the PIN tool extractTrace to extract the

[15] getMainExecutableAddress.py which can be found in the appendix
[16] Available in the appendix

function actually executed when the program runs normally. This information is the original trace for the target program with the given sample file. Finally step 2b is performed by calculating the number of hops from each of the functions recovered in step 2a to the function where the crash occurred (generated in step 1c) by traversing the call graph for the target program (extracted in 1a). From this information we can find the shortest number of hops away from the original trace where the crash occurred. Step 2 (both a and b) is automated with a script which starts the required tools for each of the fuzzing results.

First of I started extracting all the functions of the target program and the adherent call graph for these functions. For this purpose I used BinNavi with a specialized script running through all the functions. By utilizing information about child functions, it was possible to generate complete call graph information for the entire target program. The first step of this process was to open the binary file with IDA Pro for disassembly so that a disassembled database file could be generated. This file was then exported into BinNavi. The final step of this process was then simply to load the script file and make it execute with the loaded data, producing the call graph information. The script used is called getCallgraph.py[17] and is written in Python. The output is a .dot file located at the BinNavi installation directory. The content of the file is in the format: "parent_function" -> "child_function". The file is populated with one line for each possible function call in the target program. This script also outputs another file with information about the functions that belong to the main program.

7.1.1.1 Fuzzing: producing crashes

When the set of samples finally had been reduced and we had the information needed about the target program's function call graphs, it was time to get some crashes. Peach uses a pit file when cracking data from the sample file into the data model. The data model used is a very simple and dumb one, which simply defines the entire content as a binary blob. The pit file also defines a logger mechanism for recording crashes and interesting information. The number of crashes that peach discovers, directly affects how much data will be available when making a conclusion and evaluating the final results. To be certain to have enough crashes to work on, the fuzzing process needed to run for a rather long time. The command used when starting the fuzzing is as follows:

```
python peach.py pdf_semi_smart_sumatra.xml
```

Figure 8: fuzzing command

Argument	Usage/meaning
Pitfile.xml	Contains information about data model, fuzzing technique, location of samples, target program and logging information

Table 6: fuzzing command's arguments

[17] Can be found in the appendix

The pit file used operated on the minimum set previously produced. Logging was performed directly to the host via the shared folder.

7.1.1.2 Extracting the trace

After a longer fuzzing process, some crashes were discovered. For each of the unique crashes, I ran the original file which caused the crash in each incident through the PIN instrumentation tool with ExtractTrace. This tool extracts every single instruction of the target programs which were used during normal execution. In addition the PIN tool also extracts information about all instructions of the entire target program. This information was recorded in two separate text files, for later use in other analysis scripts. When using the PIN tool with GUI applications, a minor problem arose as the application typically never terminated before issuing a closing order via the GUI. This had to be done manually, or through a third party script which had to be started before launching PIN. Such a script is included in the appendix[18]. The specifics of how PIN operates can be found in its manual, but the PIN tool (extractTrace) that is used, instruments the target program at each image load and iterates through all functions and their instructions. To extract the trace of the function, each iterated instruction has an analysis routine added to it, which is executed if the instruction is called (executed), resulting in the full trace of the program. The image instrumentation is also used to iterate over each section, and all of its instructions, recording information of all instructions that exist.

7.1.1.3 Modifying the trace

As I decided to only investigate the target program's specific code, and not code from other shared or dynamically linked libraries (DLLs), a script was run after each finished instrumentation process to remove some information from the file produced by PIN and ExtractTrace. This is possible with the information gather from BinNavi (and modified by Immunity) with the file containing information of all functions belonging to the main executable and their addresses. The call for this script, named modifyExtractTraceResults.py is as follows:

```
python modifyExtractTraceResults.py functions.txt  mainExecInfo.txt
```

Figure 9: modifyExtractTraceResults command

[18] The script is called KillApp.py

Argument	Usage/meaning
functions.txt	File produced by PIN with info of all functions and all their instructions
mainExecINfo.txt	The file produced by BinNavi and Immunity containing information on the bounds of the main executable image.

Table 7: modifyExtractTraceResults command's arguments

For each crash, after completing the instrumentation for the appurtenant original sample file, a Python script was used to get the address of the function where the crash occurred. This script operates on the file generated by PIN which includes information of all functions and their instructions in the program. The script runs through each line in the file, searching for an instruction corresponding to the crash address, and if found, the corresponding function address is written to an output file. This operation was then followed by another Python script which calculates the number of hops between the original trace and the crash location, for the crash currently being investigated. This script appends its results to the file "distance_from_trace.txt". Each line in this file corresponds to an individual crash. The contents of each line are the original filename, the crash location, the closest function in the original trace and the number of hops between the crash location and the closest function. Each field is separated by a tabulator.

The calls used for the two scripts described are:

```
python getFunction_for_crashInstruction.py <crash address> functions.txt
```

Figure 10: getFunction command

Argument	Usage/meaning
Crash address	The address of the location of the crash, in hexadecimal notation
Functions.txt	File produced by PIN with info of all functions and all their instructions

Table 8: getFunction command's arguments

7.1.1.4 Finding the number of hops between the location and the trace

```
python hops_from_trace.py graph.dot trace.txt <crash function address>
```

Figure 11 : hops_from_trace command

Argument	Usage/meaning
Graph.dot	File produced by BinNavi when run with script getCallgraph.py containing all function calls in the target program
Trace.txt	File produced by PIN with info of all functions touched during normal execution
Crash function address	The address extracted from the getFunction_for_crashInstruction.py – address of the function where the crashed instruction resides.

Table 9 : hops_from_trace command's arguments

To automate the process of instrumenting and calculating the number of hops from the trace for each of the crashes discovered, a wrapped script was used. This script automatically gets the information needed for each crash, and then starts the instrumentation, the modification of the instrumentation results, the extraction of the crash function address and finally launched the script used when calculating and outputting the distance between the crash and the trace. Before commencing the work described above, the wrapper also launches another Python script called killApp.py which searches for a given process name, and if the process has been inactive for a given time, that process is killed. This functionality is needed as SumatraPDF will never terminate when launched through PIN. Therefore, to make the entire process progress, SumatraPDF needs to be terminated when it has completed all its work and the full trace has been collected by instrumentation. The killApp script takes the name of the process to kill as an argument and can therefore be utilized with other target programs as well. The wrapper script is launched as follows:

```
python crashAnalysis_wrapper.py Z:\Host
storage\peach_logtest\pdf_semi_smart_sumatra.xml_2013Nov07083057\Faults
E:\Master\pin-2.11-49306-msvc9-ia32_intel64-
windows\source\tools\MyPinTool\compile_n_run.cmd
```

Figure 12 : crashAnalaysis wrapper command

Argument	Usage/meaning
Z:\Host storage\peach_logtest\pdf_semi_smart_sumatra.xml_2013Nov07083057\ Faults	Path to faults folder produced from fuzzing
E:\Master\pin-2.11-49306-msvc9-ia32_intel64-windows\source\tools\MyPinTool\compile_n_run.cmd	path to compile_and_run.cmd file for pin execution
C:\egenkompilert_sumatrapdf\SumatraPDF.exe	path to target program

Table 10 : crashAnalaysis wrapper command's arguments

Each operation described in this section relating to the first research question, has to be repeated with the second target program. This is to get a broader set of data.

7.1.1.5 Locating the branch-off location

When working with code coverage and (code) traces it is also interesting to see the differences between the trace from a normal execution and the trace from an execution with a mutated file. In particular it is interesting to see where in the code the mutated execution "branches off" from the original trace. As creating and looking at the traces from various executions already was a rather big part of this research task, I decided to make use of some of the work that was already done. To be able investigate and discover the location where a trace of a mutated file branches off the original trace, a couple of specific scripts had to be created and run in the order which is soon to be presented. The result I wanted to get was a list of the last common location for the two traces, as well as the next address in each of the two traces. I decided to only use one target program during this process as I simply wanted to create a method to extract the needed information and see the results, and performing all operations on a second target program would only produce a new set of locations, which in fact would not add much value to the set of results since all programs will have different results. The target program used was SumatraPDF as this program contains the largest code base of the two target programs. Furthermore I randomly picked four of the samples from the minset operations. Each of these four files got mutated 10 times resulting in a total of 40 mutated files and 4 original samples. The traces for all 44 files were produced. Finally for each of the 4 original samples, the traces for the 10 appurtenant mutated samples were compared to the trace of the original samples and the desired data were extracted. Random samples were used as the strength of the picked samples in regard to achieving good fuzzing results when used in a fuzzing process is not important due to the fact that I simply wanted to uncover the branch-off location. I consider the process in itself, to be valuable if there is an interest in investigating this particular property. The operations described were performed in the following order:

1. Choose 4 samples and put them in a folder
2. Run traceAllFiles.py on the 4 samples
3. Run peach –range 0,9 with the pit file generate_samplex.xml once for each sample
4. Run getMutatedTrace.py with the traces for the mutated sample
5. For each of the 4 normal samples, run locateSplit.py to generate a file with the results

7.1.2 Results

To recap, an original trace (for a sample) is simply the first trace of the execution of a target program with one specific (non-mutated) sample. The trace for a program with a sample before mutation is referred to as an original trace, while the trace for the same program with a mutated version of the same sample is no longer an original trace.

After the fuzzing process performed in conjunction with this research task, several faults were detected. Typically each unique fault was produced by several different input files, or with several various mutation techniques for the same file, yielding many crashes for the same fault. Firstly I present the number of unique faults detected and the number of various crashes that caused the given fault.

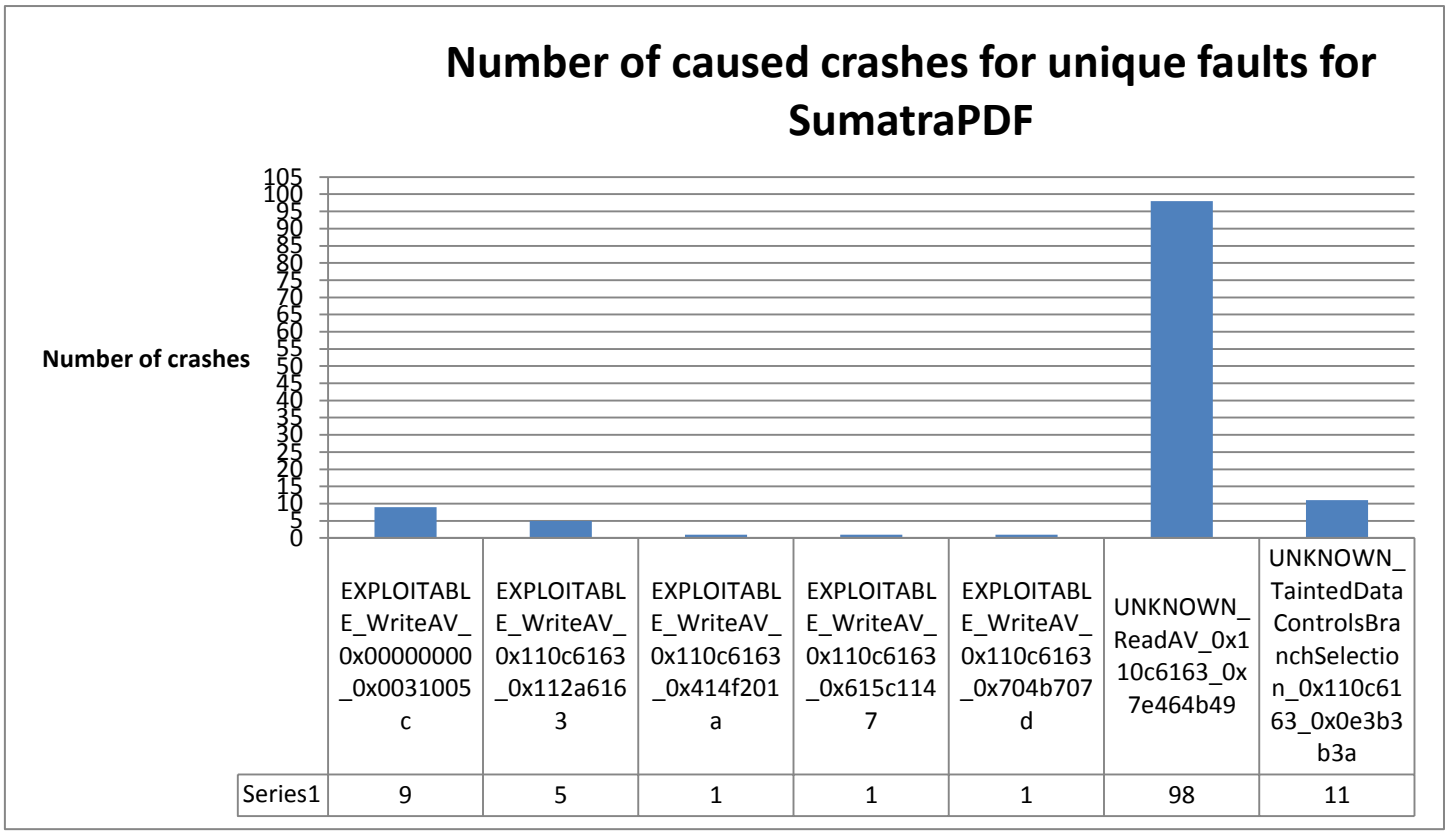


Figure 13: SumatraPDF fuzzing results

==Date of run==

Mon Nov 18 22:17:12 2013

== Run completed ==

Fri Nov 22 21:00:55 2013

After having performed step 2, as discussed in section 7.1, the results were a text file containing information of the distance, in the terms of how many functions apart, between the function where a crash occurred, and the closest function in the original trace. In addition, the address of the function where the crash occurred, and the address of the

closest function in the trace were also present in the text file. The crashed instruction address is also included in the presented results. From the crashes after the fuzzing process, the following were the results of the distance calculations:

Address of function where crash occurred	Address of closest function in the original trace	Instruction crash address	Distance between the two functions
40c07f	40c07f	40c0ac	0
40c07f	40c07f	40c0ac	0
40c07f	40c07f	430d09	0
40c07f	40c07f	40c0ac	0
40c07f	40c07f	40c0ac	0
42742f	42742f	427466	0
425424	425424	425424	0

Table 11: Distance between crash function and closest function in trace, for SumatraPDF fuzzing results

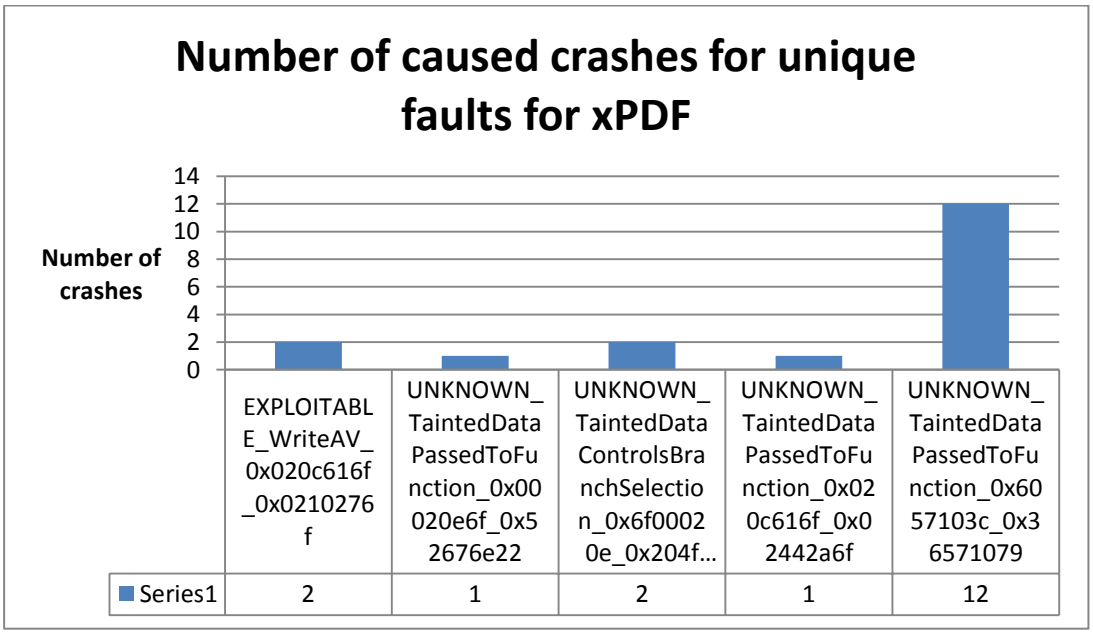


Figure 14: xPDF fuzzing results, part 1

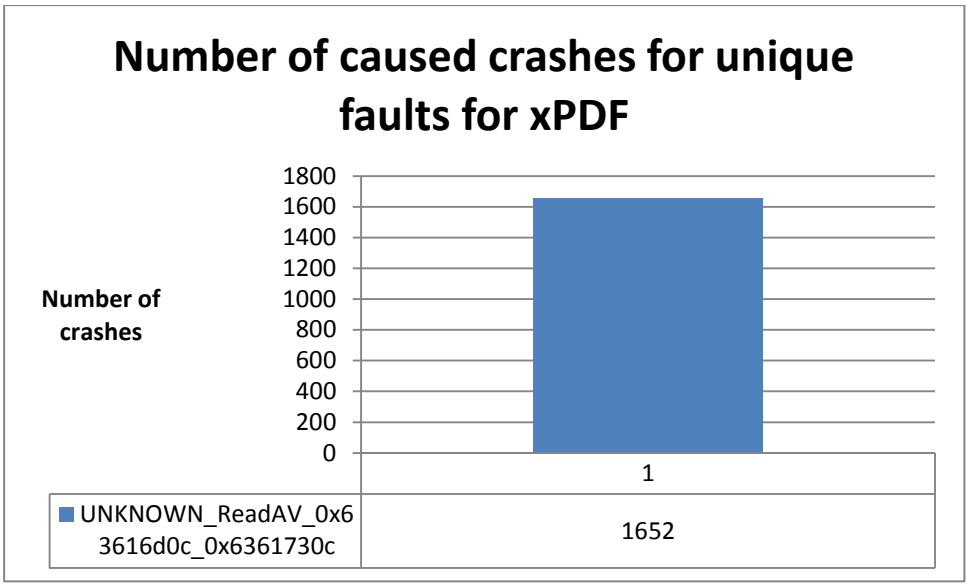


Figure 15: xPDF fuzzing results, part 2

The fuzzing process of xPDF was interrupted three times resulting in a total of 4 different fuzzing runs representing parts of the full process. Therefore there is an interval between the start and the end time that was not used for fuzzing. Regardless the start time and end time will be presented, but accompanied by the total run time.

== Date of run ===

Tue Nov 26 11:58:08 2013

== Run completed ==

Wed Dec 04 00:06:17 2013

=== Total run time ===

2 Days 17 Hours 49 Minutes 24 Seconds

After having performed step 2, as discussed in section 7.1, the results were a text file containing information of the distance, in the terms of how many functions apart, between the function where a crash occurred, and the closest function in the original trace. In addition, the address of the function where the crash occurred, and the address of the closest function in the trace were also present in the text file. The crashed instruction address will also be included in the presented results. From the crashes after the fuzzing process, the following were the results of the distance calculations:

Address of function where crash occurred	Address of closest function in the original trace	Instruction crash address	Distance between the two functions
44d108	44d108	44d397	0
416646	416646	4167cf	0
459608	459608	459615	0
459608	459608	459615	0
445490	445490	44579d	0
41a83c	41a83c	41aac1	0
4468c8	4468c8	44696c	0

Table 12: Distance between crash function and closest function in trace, for xPDF (pdftotext.exe) fuzzing results

As seen by the various crashes produced when fuzzing with peach, the various crash addresses, reasons and the level of exploitability varies to a rather great extent. This is valuable to ensure that the data is representable for the task in question as well as to ensure coverage of different parts of the target programs.

The results from the process of investigating and discovering where the trace of one program running with mutated input files differentiates itself from the trace of the same program when run with the appurtenant original sample of the mutated sample used, is a list with three locations in the code base of the program for each mutated file used as input. The first of these three addresses is the last function which both traces have in common. The last two addresses are the first function for both the original trace and of the trace of the mutated file execution after the split. The following tables contain this information for all the ten mutated files used. There will be four tables, one for each of the original samples that were picked from the results of the minset operation.

Last common address	Next address in original trace	Next address in mutated trace
539915	46807e	45f8a4
539915	46807e	45f8a4
539915	46807e	45f8a4
539915	46807e	45f8a4
539915	46807e	45f8a4
539915	46807e	45f8a4
4672c8	539915	46807e
539915	46807e	45f8a4
539915	46807e	45f8a4
539915	46807e	45f8a4

Table 13: 1st sample's trace "branch off" detection results

Original sample file name: http___4h.msue.msu.edu_uploads_files_83_4-HAnimalJudging_4HDogJudgesMay-11.pdf

Last common address	Next address in original trace	Next address in mutated trace
454349	46807e	45ede5
45f8a4	539928	46807e
454349	46807e	45ede5
539915	45f8a4	46807e
45f8a4	539928	46807e
450d2a	454349	46807e
450d2a	454349	46807e
45f8a4	539928	46807e
454349	46807e	45ede5
454349	46807e	45ede5

Table 14: 2nd sample's trace "branch off" detection results

Original sample file name: http___akademiki-akademiki.narod.ru_brain.pdf

Last common address	Next address in original trace	Next address in mutated trace
4672c8	46807e	539915
4672c8	46807e	539915
4672c8	46807e	539915
4672c8	46807e	539915
4672c8	46807e	539915
454349	46c9a3	46c54d
454349	46c9a3	46c54d
454349	46c9a3	46c54d
4672c8	46807e	539915
454349	46c9a3	46c54d

Table 15: 3rd sample's trace "branch off" detection results

Original sample file name: http___arhp.org_uploadDocs_pills_healthmatters.pdf

Last common address	Next address in original trace	Next address in mutated trace
45f8a4	46807e	45f8a4
539915	45f8a4	45f8a4
539915	45f8a4	45f8a4
45f8a4	46807e	45f8a4
45f8a4	46807e	45f8a4
539915	46807e	45f8a4
4672c8	539915	46807e
45f8a4	46807e	45f8a4
45f8a4	46807e	45f8a4
539915	46807e	45f8a4

Table 16: 4th sample's trace "branch off" detection results

Original sample file name: http___fermi.uchicago.edu_publications_PDF_oka247.pdf

7.1.3 Analysis

The results presented for this research task will be used as the basis to draw conclusions from, for the given research task. The results are not too extensive and it would certainly be beneficial to have more data to work on. However, I do believe that if given the opportunity to scale up the amount of data produced, it would be clear to see that the technique used would be both suitable and rewarding for its purpose.

When looking at the produced results for the distance between a crash and the original trace, we can see that some instruction crash addresses occur several times for a given target program, for instance address 0x40c0ac for SumatraPDF and 0x459615 for xPDF. A potential reason for getting such results could be due to the way that Peach operates. If the same kind of main mutations, potentially with small twists or some additional minor

mutations are performed in succession for the first x number of iterations, these first x number of iterations would produce fairly similar samples, which in turn would potentially trigger the same kind of crash at the same location. The samples would be mainly the same, but they would however vary a little due to the minor twists done to the main kind of mutation technique used and thus it is possible to trigger two different kinds of crashes at the same location, which is reflected by the fuzzing results figures presented.

When looking at the result of the process of finding the distance between the original trace and the function where a crash occurred in the target program SumatraPDF (Table 11), some crash instructions are listed twice. This is due to the fact that peach found several different crashes caused by the same instruction, which also possibly were caused when executing the program with different input sample files. Furthermore, it can also be seen that there are instances of two or more different crash instruction which resolved to the same function; specifically crash instruction 40c0ac and 430d09 that resolved into function 40c07f. What this means is simply that within the same function, there are two different instructions that each produced unique crashes. More interestingly is the values in the column “Distance between the two functions”. This column describes the distance between the crashed function and the closest function in the original trace. A higher number than 0 in this column reflects the fact that the crashed function is not normally reached during normal execution, but can be reached via various function calls from one of the other functions that are in the original trace. However, for all investigated crashes the distance has between the original trace and the crashed function has been calculated to be 0. This means that for all discovered crashes of SumatraPDF, the function that contained the crash instruction is in fact a function that is contained within the original trace.

Some of the same trends as described for the result of SumatraPDF above are also valid for the result of the same process for the target program xPDF. The exceptions are that no crash instruction is listed more than once, also no two non-equal crash instruction are resolved to the same function. The distance between the crashed function and the original trace is however calculated to be 0 for all investigated crashes, for the target program xPDF as well. As for SumatraPDF, this means that all functions that contain a crash instruction are indeed executed during normal execution. This could indicate that during the fuzzing run when crashes are discovered, the mutated samples used does not yield as high code coverage as desired, or it could indicate that a large portion of the code for the target program are indeed reached during normal execution with a valid sample.

When discovering where the trace with a mutated input file differentiates from the trace of the normal input file appurtenant to the mutated one, the results were four tables with data, one for each normal sampled used. In several of the tables there are entries which are identical to some other entry in the same table. This means that some of the mutated files that were used produced somewhat similar traces and thus it is possible to branch off the

original trace at the same location. This can occur if the elements that were changed in the mutation process did not affect some of the branches in the code basis, or they are set to the same value. On the other hand one can also see that there is no table where all entries are the same, which proves that the code trace does change depending on the input file, resulting in different code coverage, and touches different parts of the total code.

It can also be identified that one “next address in original trace” of one file can be seen as an entry in the “Next Address in mutated trace” for another file, which indicates a high level of complex code with branch predictions using input data, either directly or indirectly by using a value tainted by input data, while typically relying on several factors.

When opening the target program in a debugger and navigating to one of the addresses found on in the column “Last common address” for either of the files, one is typically able to find calls to the other addresses listed on that row (line), but not always as sometimes addresses on the stack or from registers are called, or other more complex mechanisms (which is impossible to interpret before runtime) are used. In some situations the current function may also simply return and the next function in the trace may be called from the parent function of the one listed in the tables.

7.2 Pre-fuzz

- *Investigate the importance of templates and their impact on fuzzing results. Adding features to the template and investigate differences in results, trace and code coverage. Learn what parts of the data model that are interesting. Determine to produce samples by creating them from a valid program or generating from a data model.*

In this research task I will differentiate between creating samples, denoting to make a sample by saving the content of an editor to a file, and generating samples which denotes to make peach produce samples from a data model/ input data file. This section deals with the second research task; Pre-fuzz.

7.2.1 Work Process

After having researched whether code coverage is important for uncovering as many faults as possible when fuzzing, the next focus is the templates used when fuzzing and how they impact the results of a fuzzing process. More specifically I investigated how various features of the file format used triggers various code segments and branches. I also tried to create a method to automatically find what feature of the sample file that gives the best code coverage when utilized with the target program. Furthermore I tried to deduct a process of detecting where in a sample file, in the context of the address of the binary file, it would be

prudent to focus mutation or alterations during fuzzing to maximize fuzzing effectiveness in the terms of faults detected as a result of increased code coverage. I investigated the previously mentioned properties for samples with sample files that were created from a program by saving the content to a valid PDF file. The same research was conducted on files produced by generating the files based on a data model.

To recap, an original trace (for a sample) is simply the first trace of the execution of a target program with one specific (non-mutated) sample. The trace for a program with a sample before mutation is referred to as an original trace, while the trace for the same program with a mutated version of the same sample is no longer an original trace.

To investigate this second research task, which as seen, has several goals some steps has to be performed for each of the goals. These steps are:

Find the features that increase code coverage the most:

1. Create samples files, one file for each feature.
2. For each sample file made, run the target program through PIN, calculating the code coverage with the current sample.
3. Calculate the difference in code coverage between the original trace and the trace of the execution for each sample file.
4. Create a sorted list of features that give the most increase in code coverage.

Discover the best locations to perform mutation in the sample:

5. Compare all the sample files with the file used when getting the original trace while recording the locations where the two files differ.
6. Count the occurrences of the various addresses from step 5, before producing a sorted descending list consisting of the location address and the occurrences

The result from step 4 gives an indication of what features of the file format that are important to include when producing valid sample files manually. The results for step 5 and 6 yield two different indicators of where in the file format it would be valuable to focus future mutations. Certainly the correctness of these results would increase as the number of samples, and unique features, increases.

7.2.1.1 Investigating the impact of various features

Step 1 was simply performed by creating samples with the use of a common text editor with the ability to save the document as a PDF file. Step 2 was performed by having a Python script, starting the appropriate sub-processes for each of the sample files created. The appropriate sub-process in this context is the PIN instrumentation tool with a custom made PIN tool called Trace which calculates the code coverage of the target program based on the instructions executed. The next two steps, 3 and 4, were completed by a Python scripts; calculateCCDiff. Step 5 was done through the 010

Editor program with a custom script called outputDifferences. The script had to be run once for each sample file to include. Finally step 6 was be done by running the python script sortDifferences.

When creating the sample files I started off with a text editor which I knew had the ability to save the document as a PDF: Microsoft Word 2010. I decided that 10 different samples, each containing a different feature would suffice for the purpose of this research task. The very first sample was simply a blank document, so that when the program was executed with this blank file, we could use the execution information as a reference when comparing information from later executions of the program with other samples. Typically some additional code has to be executed within the target program when parsing, displaying or otherwise working with a file that has some more complex content than the code base extracted from an execution with a blank file, which typically contains a lot of metadata. The samples created contain the following content:

0. blank document
1. table 2x2
2. picture (koala bear win7 sample pic)
3. page numbers (bottom normal)
4. paragraph with text
5. color some of the text
6. hyperlink
7. bullet points
8. equation
9. watermark
10. footnote

Table 17: Sample features

The features selected were chosen fairy randomly, simply aiming at getting some variations in the layout of the document, which in turn hopefully would trigger different code segments in the target program when executed with the various files. The amount of samples were chosen so that there would be enough data to draw some liable conclusions from, while at the same time the time required to create and process them would be acceptable.

The PIN tool, Trace, utilized in step 2 starts off by collecting every instruction that belongs to the target programs main image. To differentiate between instructions of other shared libraries from instructions of the main image, information of the image bounds are used. This information is accessible through a file created by BinNavi and Immunity in step 1a & 1b in the first research task, described in detail in section 7.1. During execution of the target program, the PIN tool then records each function executed and stores it in a tree structure. After ended execution the code coverage of that execution is calculated and appended to a file called coverage.txt together with information about the sample used. The information is appended, so that when the target program is run with another sample afterwards, code coverage for execution with all samples is saved, and not lost when tested with a new sample. To automatically launch PIN with all created samples, the python script traceAllFiles.py can be run. The script needs to be run from the same folder as where the samples to feed reside. For graphical target programs, killApp.py needs to be run before launching this script. killApp.py was described in section 7.1.

```
python traceAllFiles.py
E:\Dropbox\Master\practicalpin\source\tools\Trace\compile_n_run.cmd
E:\Dropbox\Master\Other\SumatraPDF.exe

NB: the mainExecInfo.txt from BinNavi and Immunity needs to be available in the root folder of PIN prior to
```

Figure 16: traceAllFiles command

Argument	Usage/meaning
compile_n_run.cmd	Path to PIN launch bat/cmd file for the given tool
SumatraPDF.exe	Path to the target executable program

Table 18: traceAllFiles command's arguments

When all samples have been fed to the target program, and their code coverage information is recorded, a Python script is run to calculate the difference between the code coverage of the sample of the blank document and the code coverage of each other recorded sample. The script used is called calculateCCDiff.py and takes the text file from the instrumentation as an argument. This same script also takes care of the work described in step 4. When the script has calculated the difference in code coverage, it can easily also arrange the entries into a sorted list. The modified data is written to the input file, overwriting the previous data.

```
python calculateCCdiff.py  coverage.txt
```

Figure 17: calculateCCDiff.py command

Argument	Usage/meaning
coverage.txt	The path to the coverage.txt file produced by PIN containing the CC for each sample

Table 19: calculateCCDiff command's argument

The steps thus far results in a sorted list showing the most rewarding features of the file format (among those tested) in the context of highest code coverage. This is interesting to see as higher code coverage potentially means more faults detected when the files are used for fuzzing purposes.

7.2.1.2 Comparing samples to find key areas to focus on

This research task does however aim at investigating some other aspects and properties of the samples used for fuzzing, and the next interesting part revolves around being able to automatically find locations in a sample file which are more important to focus on while performing mutation or editing the file. I decided that an interesting location is a location where the data often differ when

comparing a blank file with a file containing some random feature. Such locations would typically indicate a part of the file format that often has another value in more complex files, with higher code coverage, and changing the field at the given location would therefor potentially change the file in a way that makes it closer to a more complex file with higher code coverage.

In step 5, the locations where it could be valuable to focus mutations in the file format are found. To investigate this objective, 010 Editor was used to open up a blank document. Furthermore, by using a custom script for the tool, each of the other samples with different features are also, in turn, opened in the editor and compared to the sample with the blank document. Addresses where the two samples have segments that differ are printed to a file. The information is appended to the file chosen. This comparison is done in the tool 010 Editor as it has the possibility of matching regions or segments of one file to another, and figure out if parts of one file is equal to the similar part in the counterpart file, or if they differ. Some regions might also only be existent in one of the files. The 010 editor has two ways of performing the comparison:

The Comparison tool supports two different algorithms: Binary and Byte by Byte. The Byte by Byte algorithm compares corresponding bytes between the two files (e.g. each byte at address n of file A is compared only against the byte at address n of file B).
The Binary algorithm tries to identify blocks within the files that match[43].

I made use of the binary algorithm as byte-by-byte comparison would unarguably produce misleading results as the various objects of a file could vary in size and number.

After having done this process for all the used samples, the file produced by the script contained all addresses of all locations where any sample contained a different segment, when compared to the blank document. Duplicated addresses can exists in the file if more than one sample consisted of different data in the same location, when compared to the blank document. The script used through the 010 Editor is called outputDifferences.1sc. The file produced by the script can then easily be used to count the different occurrences for the various addresses where there were differences in the samples compared, and produce a new list sorted on the number of occurrences.

Counting the occurrences for each unique address in the file produced from step 5, and then sorting the addresses based on the number of times they occurred in the file, is the aim of the 6th step. These two operations are performed by yet another Python script called sortDifferences.py.

```
python sortDifferences.py  010list.txt
```

Figure 18: sortDifferences command

Argument	Usage/meaning
010list.txt	Path to the output text file after having run 010 editor with the custom script outputDifference.1sc for all samples.

Table 20: sortDifferences command's argument

All scripts mentioned in this section of the thesis, 7.2, can be found in the appendix.

Attempting to locate interesting parts of the file format in question, which as described is the aim of step 5 and 6, has been said to be done to be able to discover important parts of the file format. This in turn is important as it enables the possibility to identify and isolate parts of the file that a mutation process should focus on and parts that it should avoid altering more than necessary to easily get as good samples as possible.

7.2.1.3 Generating samples

As one of the areas of interest for this research task was to discover whether there is noticeable difference between using generated and created samples, the above described steps were repeated for generated samples, including step 5 – 7. This provided a valuable foundation for comparison of the two methods of producing samples. The process of generating samples was completed with the help of peach which is capable of generating samples from a data model or an input file. The technique of generating samples from a data model requires extensive knowledge of the file format in question and considerable time has to be put into creating a good model. When utilizing an input file for sample generation, the data of the file is cracked into the data model, which may be much more generic, for instance simply a blob. The cracked data is then modified by peach resulting in a generated sample. I chose to model the parts which were known to me and easily accessible, while using an input file to fill the rest of the data model. The input file used is the created sample produced in step 1, of the first iteration. By using such an input file, the generated sample resembles the functionality of the created sample to some extent, which in turn was favorable as these two files and their properties will be compared later. The peach pit file used for sample generation is called generate_samples.xml.

```
python peach.py -1 generate_samples.xml
```

Figure 19: sample generation command

In addition to the described research and work on this research task, I also decided to do some tests to give an even better basis for making a correct conclusion. Specifically I decided to create one more sample consisting of a combination of all features described in Table 17: Sample features, while also producing a sample by generating it based on this input file. With these two samples I calculated the total code coverage when used in the target program, to see what differences there are between using a generated and created sample which contains the same features. Additionally both files were used as samples for a fuzzing process of 3000 iterations to uncover differences in crashes produced. As exactly 3000

iterations are performed, both files have the same format, and the same pit file is used for both fuzzing runs, the exact same tests are performed and any variations in the results must come from differences in the samples themselves, as a result of two different "manufacturing" processes. In other words, the additional steps that need to be taken are:

Discover properties of generated and created samples with all features combined

7. Create one more sample with all features and generate a new sample from the created one
8. Get code coverage for both files when used as input during execution of the target program.
9. Run 3000 tests for each sample to see which produces the most crashes

For easier understanding I will recap that I differentiate between creating samples, denoting to make a sample by saving the content of an editor to a file, and generating samples which denotes to make peach produce samples from a data model/ input data file.

7.2.2 Results

For SumatraPDF which attempts to render the PDF document on the screen, a document with content that cover more of the area of the document will typically result in a grater code coverage as more code of the target program needs to be executed to render the required content. For the other target program, xPDF, which attempts to extract text from the PDF files, documents with no text (i.e. pictures and diagrams) will in theory produce less code coverage than documents containing text.

Below are the various results produced from some of the steps taken (as described in section 7.2) while investigating the objectives of this master thesis.

Results from execution with target program SumatraPDF:

The first results to be presented comes from having executed the various samples that were both generated and created, each containing one specific feature in the target program. The difference in code coverage between executing a blank document and the various documents with a specific feature has been calculated and is presented in the following two charts, both for the generated and for the created samples.

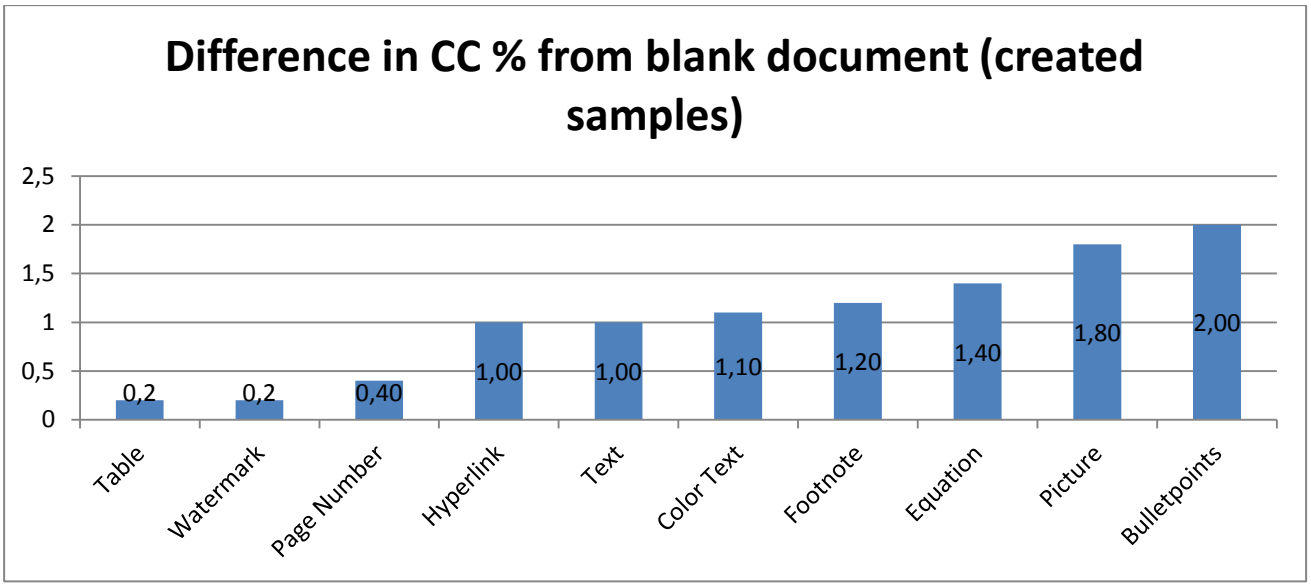


Figure 20: Differences for samples with various features compared to that of a blank sample (created samples, SumatraPDF)

Code coverage for sample with blank document: 10,8 %

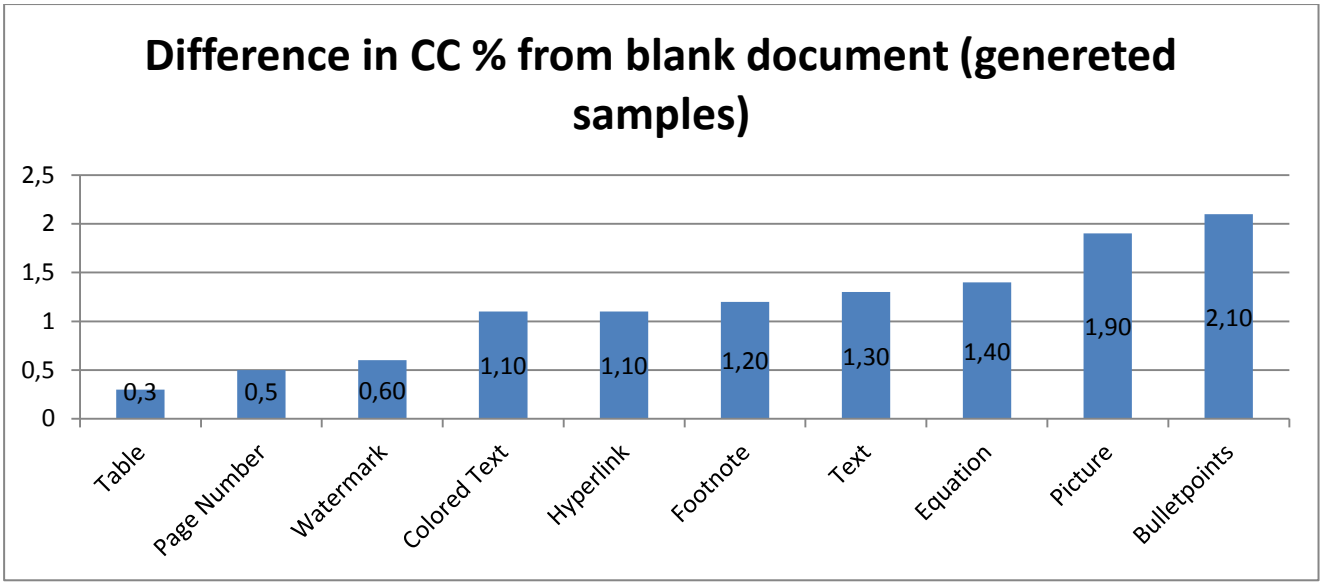


Figure 21: Differences for samples with various features compared to that of a blank sample (generated samples, SumatraPDF)

Code coverage for sample with blank document: 10,9 %

A sample consisting of all the above presented features combined into one single document was also created, followed by a generated sample consisting of the same features. These two samples were fed to the target program and their total code coverage was measured, and is presented below:

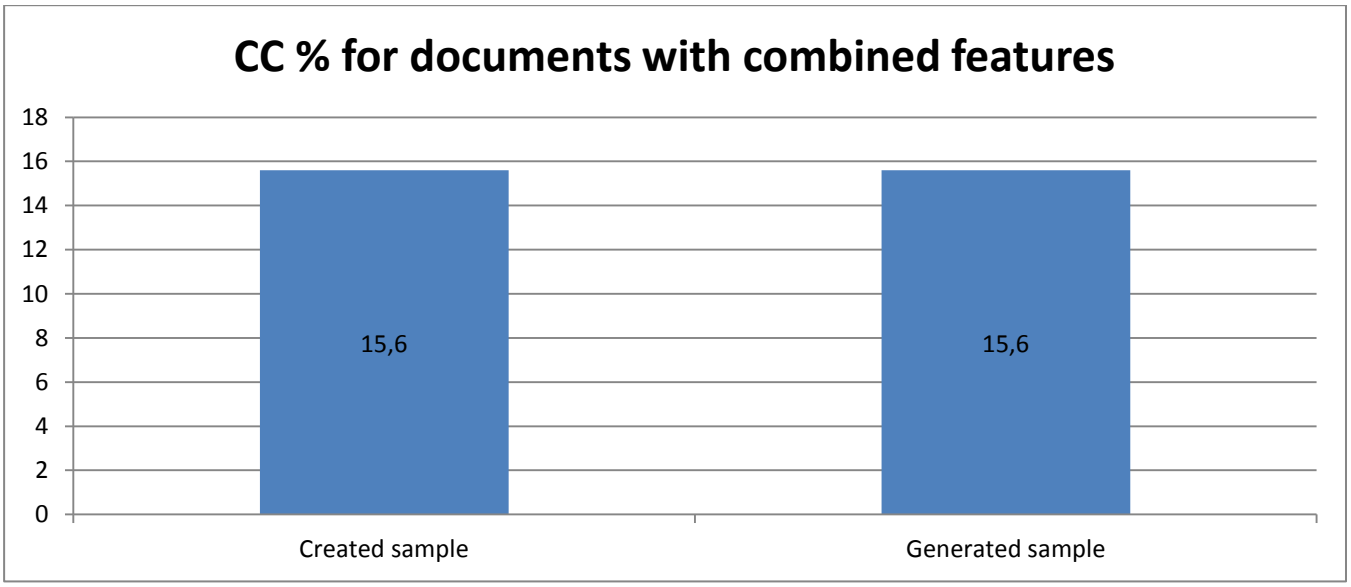


Figure 22: Code coverage for samples with combined features (SumatraPDF)

Finally the same two samples were used during a fuzzing run consisting of 3000 iterations to see if a generated and created file with the same content would produce different amount of crashes:

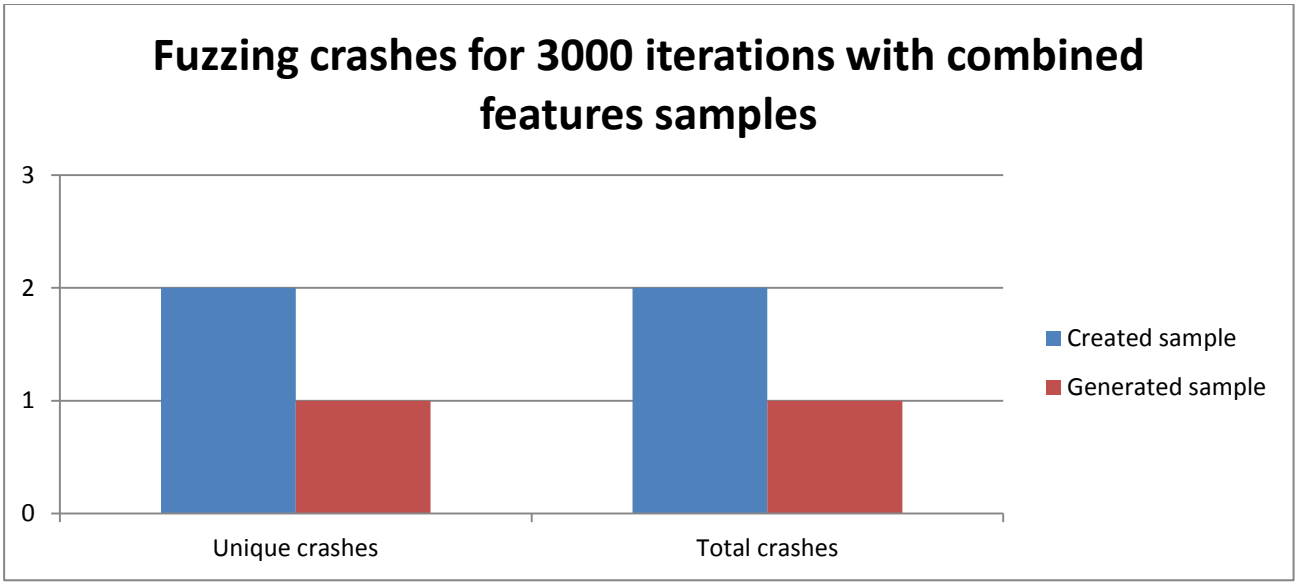


Figure 23: fuzzing results for samples with combined features (SumatraPDF)

To make sure the data is representable for the research conducted, the same tests and executions were repeated, but while utilizing the second target program xPDF, pdftotext. The appurtenant charts for the results from execution with this target program are as follows:

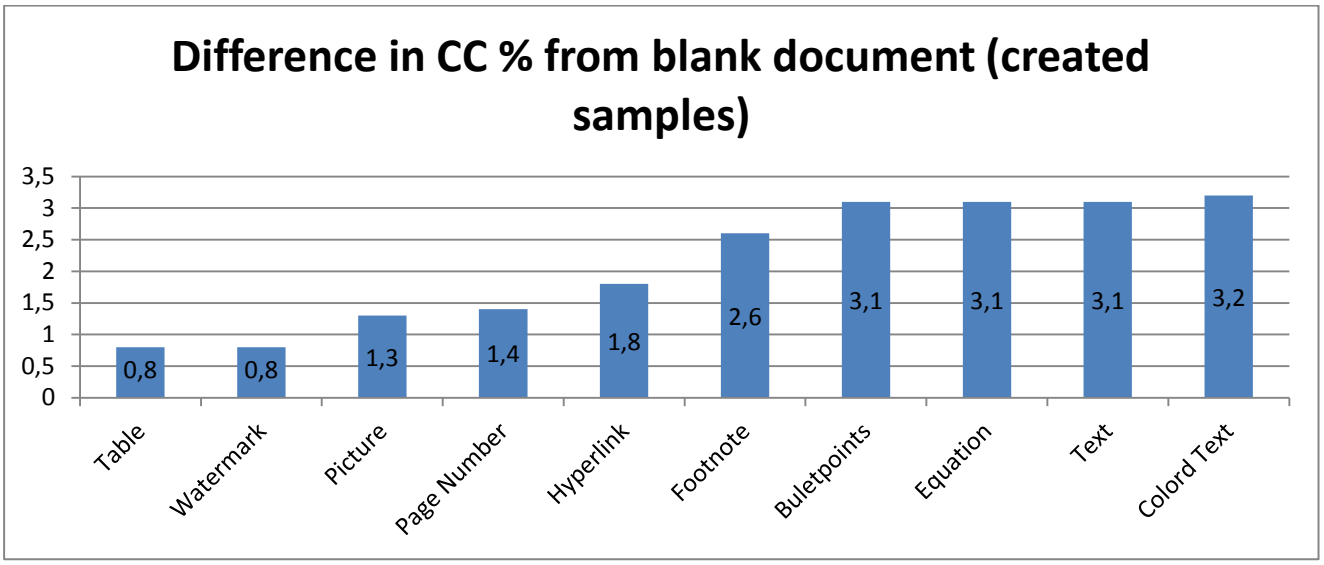


Figure 24: Differences for samples with various features compared to that of a blank sample (created samples, xPDF)

Code coverage for sample with blank document: 7 %

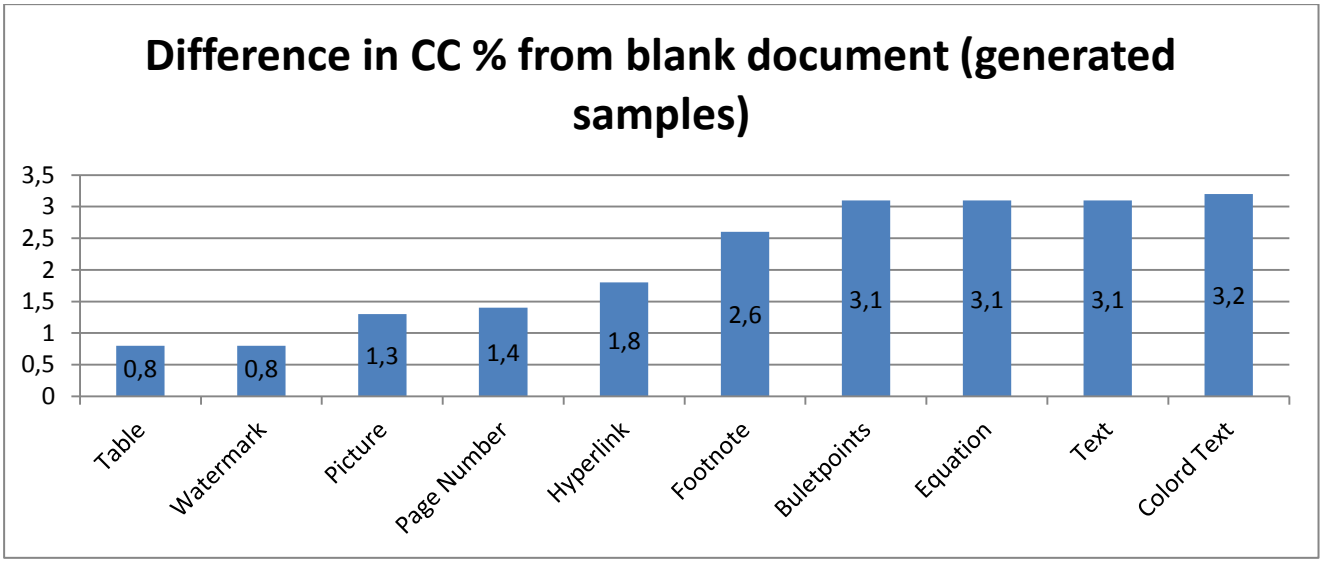


Figure 25: Differences for samples with various features compared to that of a blank sample (generated samples, xPDF)

Code coverage for sample with blank document: 6,9 %

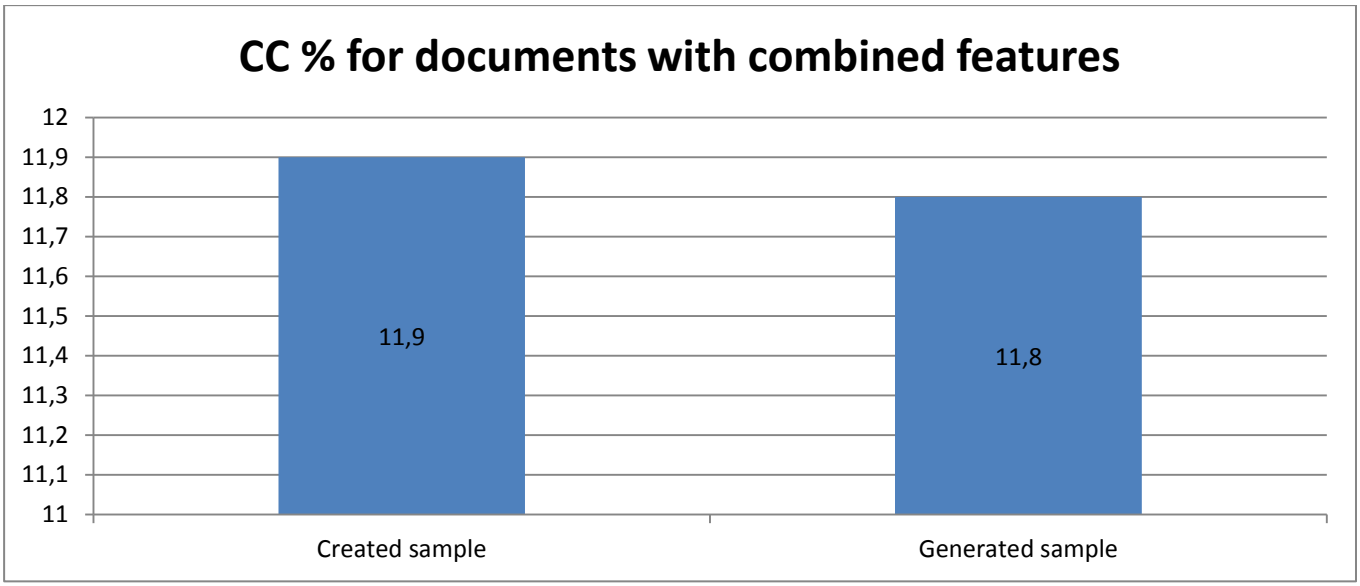


Figure 26: Code coverage for samples with combined features (xPDF)

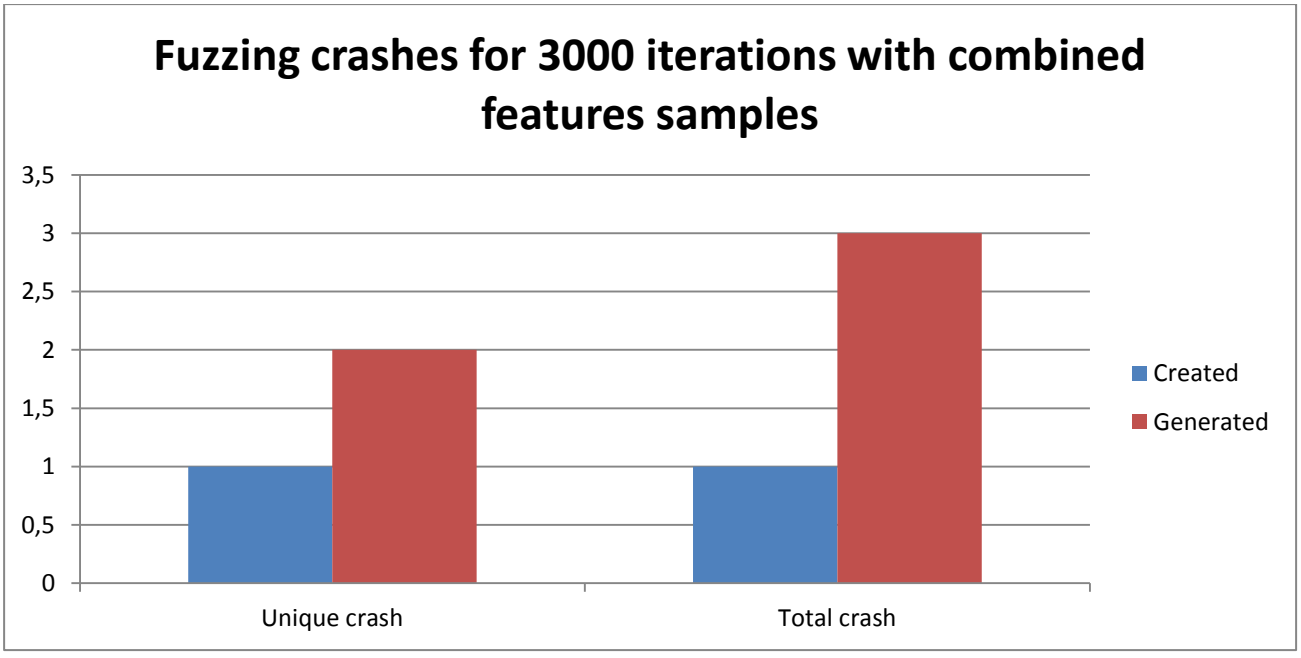


Figure 27: fuzzing results for samples with combined features (xPDF)

While the previous results of this research task are meant to investigate the difference between using samples that are generated or created, as well as to see if it is valuable to add specific addition features to the document or not, another sub-objective of this research task is to survey whether it is possible to detect specific locations in a sample that is of higher importance to focus on while producing samples. This is done by comparing several

different samples to a blank sample and see where in the file there are segments that differ. The number of samples where a given difference from the blank sample occurs is counted to see how many of the total amount of samples that contain the given difference. The ten samples with different features was compared to a blank sample to see where they differ. The chart below shows on the x-axis for how many of the ten samples a given difference (between one samples and the blank document) has occurred. As there were a total of 10 samples with different features used, the total number of occurrences cannot be higher than 10. Furthermore the y-axis describes how many unique segments in any of the samples tested, that were found in x number of samples (out of 10). For instance, the first column describes that there was only one segment (449_{16}) which differed from the content of the blank sample, which was present in all of the ten samples tested. Also the third column describes that four different segments (79312_{16}, 78647_{16}, 79072_{16} and 79467_{16}) that were different from the content of the blank sample were present in 7 of the ten files

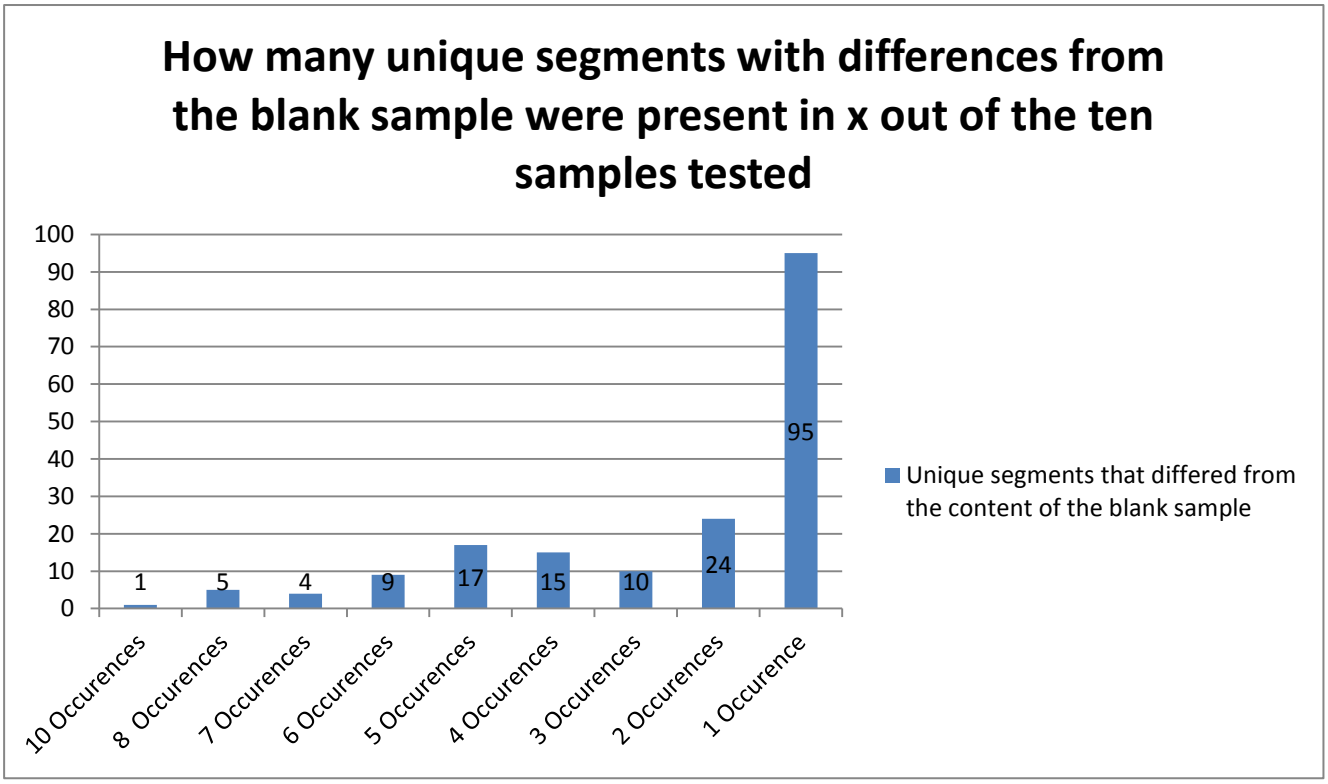


Figure 28: Number of unique locations for the various number of ocurences for differences of segments in the samples

7.2.3 Analysis

As the results from execution with the target program SumatraPDF were the first presented, these will also be the first to be analyzed. When looking at the total difference in code coverage produced by the various samples for the created samples (Figure 20: Differences for samples with various features compared to that of a blank sample (created samples, SumatraPDF)), we can see a tendency of that the more space a feature occupy in the document, the higher the code coverage is. The sample with the picture and the sample with

the bullet points, which occupy almost half a page each, have the greatest code coverage (1,8 and 2,0 percentage points difference from the blank document, respectively). On the other side of the scale are the samples containing an empty table and a watermark, both of which only differed with 0,2 percentage points from the sample with a blank document. The total code coverage of the blank document, 10,8 %, is not very high, and when the sample with the bullet points feature only yielded an increase of 2 percentage points, the maximum code coverage for the created samples run with SumatraPDF is no higher than 12,8 %. This could be explained by the fact that the samples used only consist of one single feature and the fact that the size of the documents is rather small. Larger documents spanning over several pages with a more complex compositions of features would certainly trigger more code and thus yield higher code coverage. Also, SumatraPDF is a program with a graphical user interface which is not at all used during the trace and code coverage detection process. There will typically also be a lot of fault handling functionality which are not touched during normal conditions. Based on these reasons, the rather low code coverage results can be defended. The most interesting point which can be deducted from the data is how much one given feature increase the code coverage and also how much one given feature differ from another in this aspect.

The same test that was performed for created samples was also repeated with generated samples consisting of the same feature as its created sample counterpart. The results from this process can be seen in Figure 21: Differences for samples with various features compared to that of a blank sample (generated samples, SumatraPDF). The trend described for created samples is somewhat applicable for the results of the execution of the generated samples, even though the order of features is somewhat different. There are not big differences between the created and the generated samples when looking at how much they differ from the blank document sample, but overall the generated samples have a slightly higher difference than the created samples.

As one of the reasons why the total code coverage was so low in the first execution of the samples was due to the fact that the samples used were simple, I created one more sample with all the different features combined. A generated sampled based on the created one with combined features were also produced. To see if the combination of features would increase the code coverage, these two new samples were fed to the target program while measuring the code coverage. The results of this measurement can be seen in Figure 22: Code coverage for samples with combined features (SumatraPDF). What can be seen is that there are improvements in the code coverage, almost a 50% increase. The code coverage is still not very high, but the increase in code coverage is indisputable.

As a final step, the two samples with combined features were used in a fuzzing run with 3000 iterations to see what kind of samples produced the most crashes. Only a few crashes where found due to the low level of iterations, but the results for the fuzzing of these

samples with SumatraPDF can be seen in Figure 23: fuzzing results for samples with combined features (SumatraPDF).

All of the operations performed with the target program SumatraPDF was then repeated with the second target program xPDF and the results were somewhat like those of the processes involving SumatraPDF. What is noteworthy about the xPDF results is to see that there was not one single thing that separated the results for generated and created samples, when finding the differences to the blank document from various samples (Figure 24 and Figure 25). It is also interesting to see that the order of what feature that has the most difference from the blank document is quite different for the xPDF results and the SumatraPDF results. This could be due to the fact that the programs operate differently, where xPDF aims at extracting text from the document while SumatraPDF attempts to render all the content to the screen.

For the code coverage measurement of xPDF when executed with the samples with the combined features, we can see that the created samples have slightly higher code coverage, only 0,1 percentage points. While after having performed fuzzing with xPDF we can see that the generated sample produce more crashes than the sample that was created.

The very last objective of this research task was to see if it was possible to figure out where in the sample to focus mutations. As seen in Figure 28 from the first column, there was only one location in the samples where all sample was different from the blank document (10 occurrences as a total of 10 samples were tested). Also from the rightmost column we can read that there were a total of 95 different locations where only one sample differed from the blank document. The locations will typically vary when investigating other file types as their internal structure will be different from the file type (PDF) which I have worked with in this thesis. Also, the number of occurrences will vary and depend on the number of samples used during testing, and as always, a high number of samples will produce more correct results. The purpose of my work was simply to show that there are methods of uncovering the desired information, as well as to present this method accompanied by the results for a given type of file format. The actual addresses of the location are available after finished processing, but not presented in this thesis, due to the number of different locations uncovered and the fact that they are file format dependent and thus not really interesting. If needed, they can be supplied.

7.3 Fuzzing

- *Use feedback from instrumentation to generate new fuzz cases.*

This section deals with the third and most important research task; Fuzzing.

7.3.1 Work Process

As seen, good code coverage is desirable when fuzzing to reach as much of the program as possible. As a result of this fact I wanted to develop a technique that could reach new instructions of the target program that have not been tested, as fast as possible (as few iterations as possible). The technique makes use of feedback from the current iteration to be able to make smart decisions and improve certain aspects of the fuzzing process. The feedback is provided by running the target application through the instrumentation tool PIN, and extracting various information for the execution. The decision to be made for each iteration is concerned with the strength of the current template being used during the fuzzing process. The basic idea is that if a mutated template gives better results than its original counterpart, this mutated template should be used as the new main template and further mutations should be produced based on this template. This comparison and template swap process is repeated for the newly assigned current template, until a new, better sample is created by mutation or the total number of iterations is reached. This should result in using the best possible template with regards to code coverage at the end of the process.

7.3.1.1 Getting baselines for the required time

To be able to collect feedback for each iteration of the fuzzing process, instrumentation is used. As seen, instrumentation can prolong the required execution time of an application, and to be able to extract the trace and code coverage, extra code needed to be executed for each unique basic block in the target process. This operation certainly increases the required execution time drastically. Firstly I investigated whether it was possible to perform the operations described in the previous paragraph due to the extra amount of time needed for the feedback to be able to make smart decisions. If the extra time needed for instrumentation is too great, it might be more rewarding to do several iterations of dumb-fuzzing seeing that one potentially could do many normal fuzzing iterations in the time it takes to do one iteration with instrumentation. Therefore, the first matter of business for the research of this research task was to get two baselines for the time required to run normal fuzzing and the time required for fuzzing with instrumentation.

Both timing procedures use the same seed for the fuzzing part to make sure the same mutations are performed in the same order ensuring a deterministic behavior. To time the elapsed time of the normal fuzzing run, the “Measure-Command” of Microsoft’s PowerShell was utilized. This tool starts the given command and returns the elapsed time at the end of the given command. The command used for timing the fuzzing is as follows (has to be run from within the windows Power Shell:

```
Measure-Command { E:\Dropbox\Master\Practical\peach-3.1.53-win-x86-release\Peach.exe --seed 31337 --range 0,1264499 pdf_semi_smart_sumatra.xml }
```

Figure 29: Time measure command for normal fuzzing

When starting the instrumentation after each fuzzing iteration, some problems arose regarding starting the instrumentation process through the Peach fuzzer. Finally after much testing I decided to create an external Python script managing the work flow and calling the appropriate sub processes. The tool opens the folder of the samples and runs a user supplied number of mutations on each file by feeding the file to peach. Together with the same seed for previous iterations to ensure deterministic behavior, the –range options is given to peach to make sure the next planned mutation is performed. By giving two equal numbers to the –range option I made sure that only one iteration/mutation was performed per peach call. After the call to peach has ended, a call to PIN with the target executable as well as the newly created fuzzed file from peach is supplied. Between, before and after the various calls some move, copy, remove and rename operations are performed on the files used. To be able to measure the time used for the fuzzing process combined with instrumentation, the peach tool utilizes a module called timeit, which measure the time for the peach call and the PIN call, and adds these two values to the total time spent for each iteration. There is some time lost due to overhead when performing sub-process calls like the calls to start peach and PIN, but this time is almost negligible compared to the time spent by the actual processes of the call, and is acceptable for the granularity needed in this scenario. The file operations performed by the python tool were not measured which is positive. The command used for timing the fuzzing + instrumentation is as follows:

```
python timeit_instrumentation.py 1500 E:\dropbox\Master\Practical\peach-3.1.53-win-x86-release\Peach.exe E:\dropbox\Master\Practical\pin-2.11-49306-msvc9-ia32_intel64-windows\source\tools\traceBBL\compile_n_run.cmd E:\Dropbox\Master\Other\SumatraPDF.exe
```

Figure 30: Time measurement command for fuzzing with instrumentation

Argument	Usage/meaning
1500	Range for each sample (should equal switchCount in Pit file).
E:\dropbox\Master\Practical\peach-3.1.53-win-x86-release\Peach.exe	Path to peach executable
E:\dropbox\Master\Practical\pin-2.11-49306-msvc9-ia32_intel64-windows\source\tools\traceBBL\compile_n_run.cmd	Path to PIN launch bat/cmd file
E:\Dropbox\Master\Other\SumatraPDF.exe	Path to target executable program

Table 21: Description of arguments for time measurement command for fuzzing with instrumentation

To make a representative number of mutations for each sample, in total 1500 iterations per sample were executed. This number can be found in the switchCount filed of the Pit file used when fuzzing. When performing 1500 iterations for all the 842 samples of the minset, a total of 1263000 iterations are executed. The results from measuring the elapsed time for all these iterations would be highly representative. However, due to the fact that the process of instrumenting every single basic block for each iteration might drastically affect the time needed, the time measurement process described may have to be aborted prematurely. If this is the case, the time elapsed for the given number of iterations of one measurement process (fuzzing with instrumentation) is simply compared with the time needed for the same number of iterations for the other measurement process(just fuzzing). As these measurements simply are done to be able to compare the two techniques, one is still able to extract valuable results and information when comparing the elapsed time, even though not all iterations are completed.

7.3.1.2 Identifying the three most interesting samples

In addition to gathering the two baselines for the time measurements, this research task's main focus remained to be investigated. To see if it is possible to make smart decisions regarding what template to use as a base for mutations several things needed to be done:

1. Run all samples of the minimum set, without any form of mutation, through PIN to get information about the CC and their traces.
2. Run evaluateTraces.py which opens the trace-files made by PIN to find the one file with the greatest CC and the two files which have the most different traces
3. For each of the tree samples found in step 2, run one instance of smart_fuzzing.py

Step 1 was simply performed by running a Python script called traceAllFiles.py from the minset folder, calling PIN with the PIN Tool TraceBBLs, resulting in a text file with the name of the input sample containing firstly the total code coverage of the execution based on the number of unique basic blocks executed, as well as the total number of basic blocks. The rest of the file produced by PIN consisted of the addresses of the various basic blocks that were executed. The list of basic blocks is sorted in an ascending order with only unique entries. The sorted list is easily generated as all BBL addresses are stored in a B-tree, and the three is traversed at the end of the instrumentation printing out the data to file.

To goal of performing step 2 is to be able to find the best samples of all the samples in the minset. This is done to find the most applicable samples to use as a starting point when launching the smart fuzz process. The script compares each of the trace & code coverage information files produced by pin with all other information files for the other samples, and finds the file with the greatest code coverage. Furthermore it also stores the trace for each sample, compares it to the trace of all the other samples and finds the two traces that are most different. This is done by calculating the total number of addresses that are in one of

the two files, but not in the other. Due to the immense number of calculations that have to be done by this script I decided to reduce the set of information files from PIN from 842 to 705. The files that were removed were the files with lowest size as these theoretically should have less unique BBLs in their trace than the remaining files (due to the fact that they contain less data, which in turn should have a lower chance of triggering different parts of the target program), so their removal should impact the results the least. This reduction greatly impacted the speed of evaluating all of the traces and enabled me to have the remaining time needed for further operations. The command for evaluating the samples is as follows:

```
python E:\Dropbox\Master\Practical\Code\3_Fuzz\evaluateTraces.py

NB! The current working directory must be the directory of the samples to investigate.
```

Figure 31: evaluteTraces command

7.3.1.3 Developing a smart-fuzz tool

The third and final step where the actual smart decisions are made and feedback from instrumentation are used to gather the information needed to make such decisions. The whole process is wrapped in a Python script which among paths to the various tools it relies on, takes as a parameter the path to the first sample file which is used as a starting base. As follows is an overview of the smart-fuzz tool:

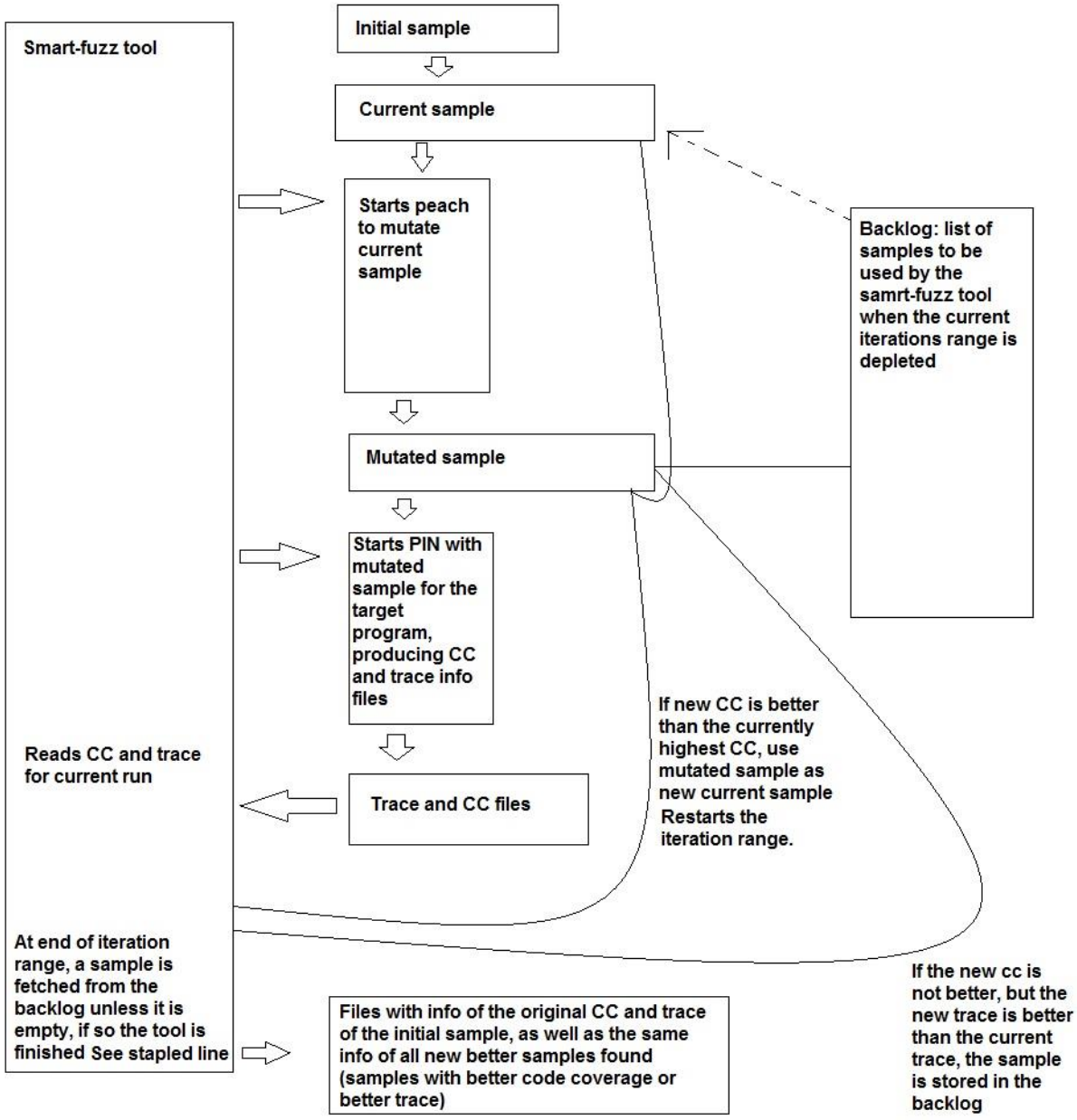


Figure 32: Smart-fuzz tool overview

The script starts off by performing a mutation on the current base sample. For each iteration a new mutation is performed on the current sample until a user specified number of iterations has been performed. When the total number of iterations are executed the script is done. However a lot more is happening during each iteration in the script. After each round of mutation, the given target program is executed through PIN with the mutated file as the input file. The PIN tool which is executed is the TraceBBLs tool which extracts and stores information about the code coverage and the trace for the last execution. Before the script starts a new iteration, the information file from PIN is loaded and read. The code coverage of the execution with the current sample is compared to the code coverage of the execution with the latest mutated sample. If the new code coverage has increased by a predefined threshold value compared to the current code coverage, the mutated sample is

considered to be better than the current sample. This threshold value should be set to a low number to include as many new samples as possible. The result of this decision is that the currently mutated sample is set to be the new base sample and the fuzzing process starts from the beginning. A threshold value is used as files with almost equal code coverage might not yield much benefit in contrast to the additional time needed to restart the mutation range. By using a threshold value, only samples with a significant increase are selected.

However, as seen, code coverage may give an incomplete picture of the code paths taken when focusing on reaching as much of the code as possible. Therefore, and additional test is performed if it is decided that a given mutated file is not better than the current base sample with regard to the code coverage. This second test is not performed if the mutated sample is seen as better than the current base sample based on the code coverage information. The focus of the second test is comparing the trace with the actual addresses that have been executed for the two executions in question (for the current base sample and the new mutated sample). Like the code coverage comparison, this test also has a predefined threshold value and if the differences between the two traces(the number of different BBL addresses) are greater than this threshold, the new mutated sample is considered better than the current base sample. However as this new mutated sample covers a very different part of the code base and has lower code coverage than the current base sample, it is stored in a log for later execution when the current base sample has finished all its iterations (NB. The current base sample may be changed due to a new mutated sample with greater code coverage before it completes its loop). Several mutated samples may be stored in the log for later execution.

After one current base sample has completed all its iterations without being changed, the log is checked for stored samples and if found one is picked and set as the new current base sample and a new loop of mutation iterations is started. During this loop new samples may be added to the log. The sample that gets picked from the log is decided by the os.listdir() Python command, which is sufficient as the order they are picked does not affect the end result.

When there are no more samples in the log and all samples have completed their mutation iteration loop range, the Python script, and the tool has completed. The results of this tool is information about code coverage and the trace length of the last sample that was used as the base sample (i.e. the only current base sample that did not get changed due to another mutated sample having higher code coverage). If samples are stored to the log during execution of the tool, information about the last sample to complete the mutation iteration range which dates from each of these logged samples also becomes a part of the final results of this tool. In practice there may be code coverage and trace information for several samples that make up the final results when the smart fuzz tool is executed. These results can be compared against the corresponding info for the file that was given to the tool as the

original base sample to see if there has been any form for improvement. The command used to execute the smart fuzz tool for the three samples is as follows:

```
python E:\dropbox\Master\Practical\Code\3_Fuzz\_smart_fuzzing.py 280
E:\Dropbox\Master\practical\_peach-3.1.53-win-x86-release\Peach.exe
E:\Dropbox\Master\practical\pin-2.11-49306-msvc9-ia32_intel64-
windows\source\tools\traceBBL\compile_n_run.cmd
E:\Dropbox\Master\Other\SumatraPDF.exe E:\pdffile.pdf
```

Figure 33: Command for smart-fuzz tool

Argument	Usage/meaning
280	range for each sample
E:\Dropbox\Master\practical\ peach-3.1.53-win-x86-release\Peach.exe	Path to Peach engine
E:\dropbox\Master\Practical\pin-2.11-49306-msvc9-ia32_intel64-windows\source\tools\traceBBL\compile_n_run.cmd	Path to PIN launch bat/cmd file
E:\Dropbox\Master\Other\SumatraPDF.exe	Path to target executable program
E:\pdffile.pdf	Path to the initial sample file to use

Table 22: Description of arguments for the smart-fuzz tool

The input value given when the tool was executed for the three resulting samples from step 2 for the number of iterations was 280. This value is much lower than 1500which is the number of iterations (reflected by switchCount) used during normal fuzzing. The reason for this decision is simply the fact that the instrumentation part of the process adds too much time to the total process. Running with a higher range (switchCount) would probably lead to the process not being able to finish within a reasonable timeframe. However, there is no doubt that the results would be better and there is a higher probability of reaching more code with more iterations. A higher range should therefore be given if the time available allows it.

The operations described in this section relating to research task 3: Fuzzing should also be repeated with the second target program xPDF, but due to lack of time/available computing power the operations were only conducted with SumatraPDF as the target program. Even though it would be desirable to run the same operations against xPDF as well, the yielded results from using SumatraPDF is sufficient for my needs as the target program should prove to be sufficient in the context of evaluating the smart-fuzz technique developed and give some interesting numbers and indications.

7.3.2 Results

Firstly the most important result from this research task is the baselines with the measured time for the two different processes; fuzzing and fuzzing in combination with instrumentation. This is important to see if it is at all feasible to instrument every fuzz case

and if the extra time is worth the gains as opposed to running many more normal fuzzing iterations without instrumentation. The results for the time measurements are as follows:

Normal fuzzing:

1000 iterations	2 Hours 38 minutes 17.196 second
19769 iterations	11 days 14 Hours 54 Minutes 38,286 seconds

Table 23: Normal fuzzing time measurements

Unfortunately the fuzzing process kept crashing at around 20 000 iterations due to "lack of resources (disk space)". My first though was that this was caused by the huge fuzz files created, but when changing the storage location to a medium with 500 gigabytes of free space and the problem persisted I was unable to solve the problem. Several attempts were made, but none reached the final number of iterations (1264500) and the furthest the process got before crashing was 19769 iterations.

Fuzzing with instrumentation:

1000 iterations	5 days 39 minutes and 51.234 seconds
4500 iterations	17 days 5 hours 14 minutes and 4.198 seconds

Table 24: Fuzzing with instrumentation time measurements

To be able to easier compare the results, the results for 4500 iterations will be extrapolated to 19750 iterations. However, keep in mind that extrapolation introduces uncertainty and is solely based on the time measurements recorded as far as 4500 iterations in this situation.

19750 iterations	75 days 13 hours 38 minutes 25 seconds

Table 25: Fuzzing with instrumentation time measurements, extrapolated

The second interesting part of this thesis is the created tool which is used to perform smart fuzzing in the context of adapting the template used for mutation to newly improved templates created during the fuzzing process. The results from this operation is the functioning tool itself, but more importantly the number of "better" templates that were generated and the code coverage and trace belonging to each of these templates. The smart-fuzz tool was used on three different base templates:

1. The sample with the biggest initial code coverage
2. One of the two samples with the most different traces
3. The second of the two samples with the most different traces

These three samples were found by running a script on the trace information for all the samples in the set, comparing the data and selecting the three files that matched the criteria specified above. The three files found were:

With a code coverage of 78,2 %:

1. http___knob.planet.ee_kirjandus_manuaalid_tv_salora3321_Nokia+4223_5123_5523_Oscar_Pal+Secam_Salora+3321U_A3-A14+Ch.pdf

And with a difference of 11798 BBLs:

2. http___texarkana.be_Chocolate.pdf
3. http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf

The results after running each of the three samples through the smart-fuzz tool will now be presented, including the data for the initial sample to be able to make a comparison. The threshold values used to determine when a new good sample has been found were set to 1.4 difference in the code coverage and 500 difference of the BBLs in the trace. A "good" sample is a sample that the tool has found to be better than the original sample fed to the tool from the user at startup. A sample is considered to be good if its code coverage is over the set threshold value, compared the previously run sample. The same goes for the uniqueness (number of differences) in the traces produced.

Firstly the information for the sample http___texarkana.be_Chocolate.pdf will be presented, starting with the code coverage and trace size for the original sample.

Code Coverage	77.2
Number of BLLs in trace	4315

Table 26: Trace and CC information for original sample of http___texarkana.be_Chocolate.pdf

Results after running the smart-fuzz tool with 300 iterations:

Number of new good samples found: 1

Code Coverage	78.6
Number of BLLs in trace	4315

Table 27: Trace and CC information for the new samples found for of http___texarkana.be_Chocolate.pdf during smart-fuzzing with 300

The smart fuzz process was repeated on the same original sample with 1500 iterations to see how an increase in the total number of iterations would impact the results.

Results after running the smart-fuzz tool with 1500 iterations:

Number of new good samples found: 1

Code Coverage	78.6
Number of BLLs in trace	4315

Table 28: Trace and CC information for the new samples found for of http___texarkana.be_Chocolate.pdf during smart-fuzzing with 1500 iterations

Secondly, the information for the sample which was found to be the most different from the http___texarkana.be_Chocolate.pdf will be presented, starting with the code coverage and trace size for the original sample. The other sample in question is http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf

Code Coverage	76.4
Number of BLLs in trace	4315

Table 29: Trace and CC information for original sample of http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf

Results after running the smart-fuzz tool with 300 iterations:

Number of new good samples found: 1

Code Coverage	78.5
Number of BLLs in trace	4315

Table 30: Trace and CC information for the new samples found for of http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf during smart-fuzzing with 300

The smart fuzz process was repeated on the same original sample with 1500 iterations to see how an increase in the total number of iterations would impact the results.

Results after running the smart-fuzz tool with 1500 iterations:

Number of new good samples found: 1

Code Coverage	78.5
Number of BLLs in trace	4330

Table 31: Trace and CC information for the new samples found for of http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf during smart-fuzzing with 1500 iterations

The last of the three samples used as the original sample when running the smart fuzzing tool is the sample that was found to have the greatest code coverage after evaluating all samples from the minset; http___knob.planet.ee_kirjandus_manuaalid_tv_salora3321_Nokia4223_5123_5523_Oscar_PalSecam_Salora3321U_A3A14Ch.pdf. This sample was also the biggest of all the samples and potentially due to the size of the original sample, the mutated files consumed all available free space, resulting in Peach crashing and the Python wrapper waiting for user input to continue. As a result that entire process consumed too much time and had to be aborted. Solving the problem of Peach producing huge mutated files would solve the problem, and is an important point of the proposed future work section.

In addition to the presented and explained operations and results I also performed some minor additional research to get some more data for this research task. Firstly I took the smallest sample from the minset (as I suspect this would be the way to fastest complete the computations required) and recorded the increase in CC per iteration, the time for each iteration and the total time, when used in normal fuzzing and with the smart-fuzz tool. The results from this investigation are as follows:

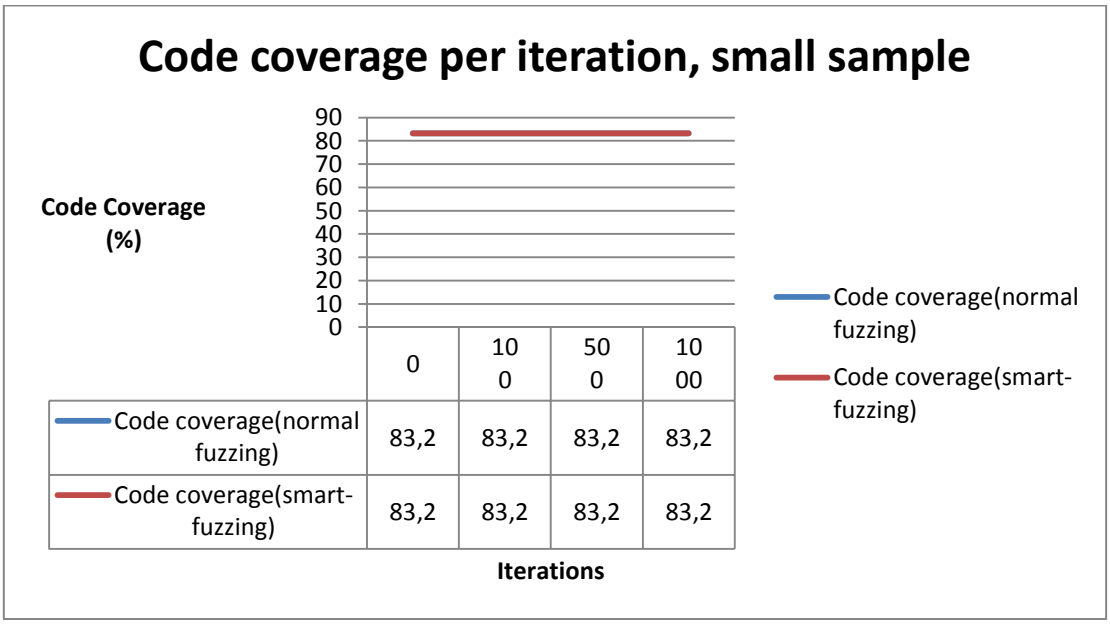


Figure 34: Code coverage per iteration, small sample

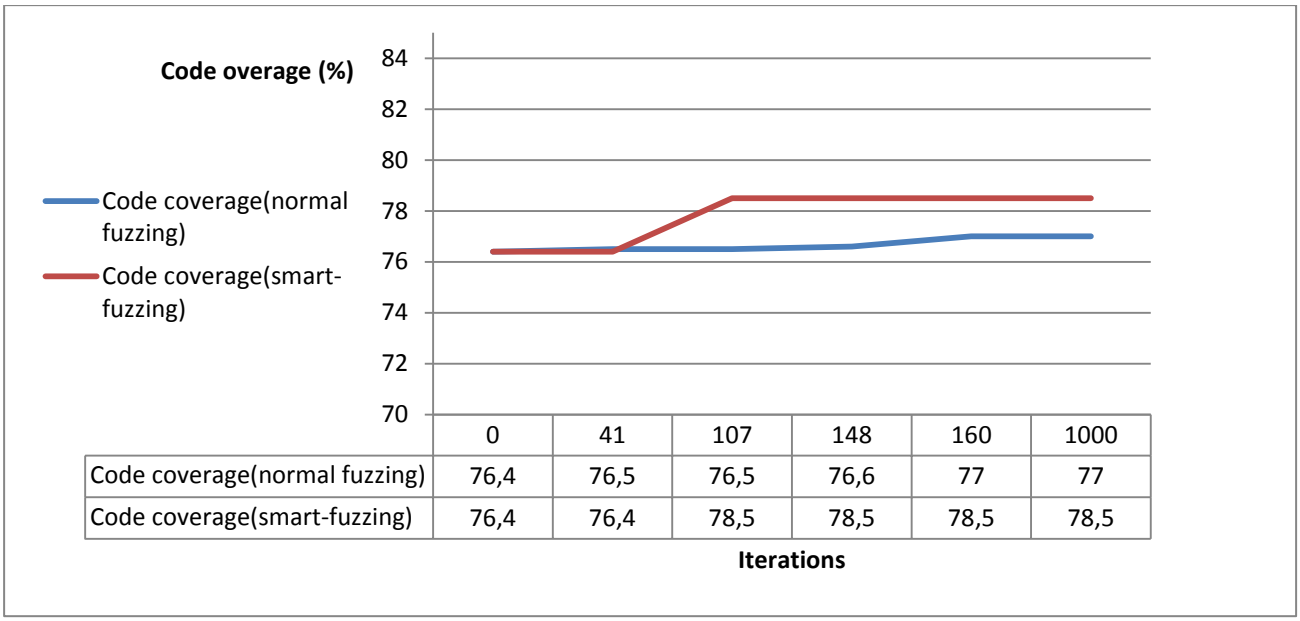
The time per iteration for the normal fuzzing with the small sample ranged from 6,77 seconds (first iteration) to 13,4 seconds (718th iteration). The average time per iteration is 11,21 seconds.

Normal fuzzing, small sample, total time	11213 seconds
Smart fuzzing, small sample, total time	103659.57 seconds

Table 32: Normal vs. smart fuzzing time requirement, small sample

The time per iteration for the smart-fuzzing with the small sample ranged from 157.64 seconds (459th iteration) to 193.6 seconds (122nd iteration). The average time per iteration is 103,66 seconds.

The results from running the smart-fuzz tool on the sample http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf during smart-fuzzing with 1000 iterations was also compared with the results from running a normal fuzzing run on the same sample for the same amount of iterations



	0	41	107	148	160	1000
Code coverage(normal fuzzing)	76,4	76,5	76,5	76,6	77	77
Code coverage(smart-fuzzing)	76,4	76,4	78,5	78,5	78,5	78,5

Figure 35: Code coverage per iteration, http___tourism.visitcalifornia.com_media_uploads_files_editor_Workplans_2012_JAPAN.pdf

As can be seen the smart-fuzzer yielded a somewhat better final sample than the normal fuzzing process. The smart-fuzzing took on total 225516,15 seconds to complete while the normal fuzzing required 169351,42 seconds to complete.

The second, and last, minor additional research that was performed in conjunction with this third research task was to attempt to find a technique to optimize the instrumentation process. I attempted to record all BBL that had been instrumented by each and all instrumentation iterations performed by one run with the same target, to be able to only instrument new BBLs for the remaining iterations of the given run. Theoretically this should improve the time required by instrumentation. This can be done as a given basic block will not change from one execution to another, and thus no data is lost when not instrumenting a given basic block twice, or more. However, the way to produce the trace and the code coverage has to be changed so that previously instrumented basic blocks that should have been instrumented in the current execution with normal instrumentation is included when calculating the trace and code coverage. Below is the timing results for running 10 iterations, each with a new sample, for the same target program, in succession using both the normal, brute force way of instrumenting (instrumenting all execution units touched each iteration), as well as the timings for the smart way of instrumenting with only new execution units.

Iteration	Current iteration time (seconds)	Total time elapsed (seconds)
1	167,2	167,2
2	33,8	201
3	49,8	250,8
4	66	316,84
5	43	359,85
6	33,3	393,2
7	34,5	427,7
8	34,5	462,3
9	36,4	498,6
10	45	543,9

Table 33: Instrumentation timings 10 iterations, smart instrumentation

Iteration	Current iteration time (seconds)	Total time elapsed (seconds)
1	168,8	168,8
2	175,4	344,3
3	174,4	518,7
4	197,85	716,6
5	182,94	899,5
6	163,88	1063,4
7	174,73	1238,17
8	170,4	1408,59
9	174,53	1583,12
10	180,81	1763,93

Table 34: Instrumentation timings 10 iterations, normal instrumentation

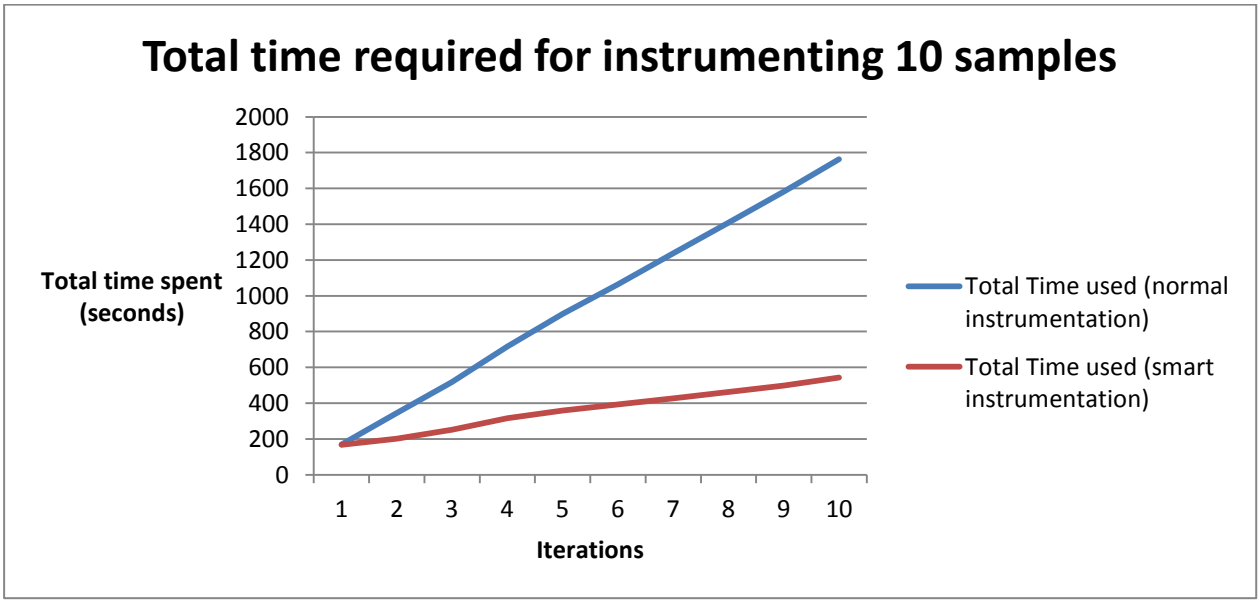


Figure 36: Total time spent instrumenting 10 samples, normal and smart instrumentation

7.3.3 Analysis

The results presented for this research task will be used as the basis to draw conclusions from, for the given research task. The results are not too extensive and it would certainly be beneficial to have more data to work on. However, I do believe that if given the opportunity to scale up the amount of data produced, it would be clear to see that the technique used would be both suitable and rewarding for its purpose.

When performing mutations based on this sample, the fuzzing tool used sometimes created extremely huge files (up to 450 GB) consuming all the free space on the selected storage medium. When Peach had used up all the needed space and still needed more, it would crash. The fact that this kind of crash was caused several time when mutating the last and biggest of the three samples could indicate that the original size of the sample may have something to do with when these huge mutated files are created and could be an interesting topic for future work. These crashes were handled to some extent by the python wrapper used, by simply skipping the iteration and telling peach to start the next iterations. However as a result of this behavior, the final printout was not created correctly and no information about the CC or the size of the trace for the new good samples discovered was printed. The number of new good samples discovered was however available. 4 new good samples were discovered when using the sample with the greatest CC (while also being the biggest) as the original base for mutations.

As mentioned, what was discovered during many iterations of the fuzzing process was that the hard disk wait time had quite the impact on the total time needed. For certain iterations Peach would produce files up to many (hundreds of) GB. The process of both creating these templates and then reading them when feeding them to the target program was rather time consuming. The cause of these huge templates is unknown, but could be an aim for further work.

The two time measurement techniques, with and without instrumentation have been done on the exact same systems; two identical virtual machines, both using the same underlying originally idle multicore CPU. As each VM also have several cores at its disposal, and the executed sub process nearly constantly uses 100% of one of the available cores at any time while the other core(s) never are fully utilized, the time estimates can be compared without too much uncertainty. The total time may (and probably will) however vary from the required time for other systems, but can nevertheless be used as an indication of the time needed.

Measuring the CPU time is the most accurate approach. However, the python timeit module used when measuring the time elapsed for instrumentation and fuzzing only measures the total elapsed time. The Measure-Command used to measure the fuzzing operations also outputs the real time used. Even though this measurement is prone to be inaccurate, the various aspects that affect the timing results are somewhat equal for both measurement techniques and the results can therefore be used to compare the two different executions

even though the actual number is not realistic in terms of pure CPU run time. The results gathered can however, as stated, be used as an indication of how much time is needed for the two operations.

When considering if it is worth the extra time to do instrumentation to get feedback, one also have to take into account the time used for calculations based on handling the feedback data, such as comparing two traces, and also file IO to get the data from the instrumentation into the program. The python script for the smart-fuzzing tool does measure the time spent on comparing the trace of the current iteration with the ones stored. It is important to be aware of the additional time for performing the operations.

As seen, the normal fuzzing run never reached to final number of iterations that were planned, but it is nevertheless possible to use the results for the purpose we needed them; to see how much difference there are of performing normal fuzzing and fuzzing with instrumentation. This is possible as the results are used as an indication of how long each process will take. Even though the results for the total planned number of iteration would be more representative, there is nothing wrong with the time measured for the actual iterations completed.

As there exist time measurement for both normal fuzzing and fuzzing with instrumentation for 1000 iterations, these two values can be used to compare the difference of the time required directly. The fuzzing process is roughly 45 times faster than performing instrumentation with the given PIN tool, based on the numbers for 1000 iterations. 1000 iterations is however not very much in the context of fuzzing and mutations and even though their appurtenant time measurements can be used for comparison, measurements for a greater number of iterations would be more representative. As seen the time measurement process never completed for either of the two processes for various reasons. To be able to utilize the recorded data for the time measurement process, I will calculate the average elapsed time for each iteration and then compare the two average values for the two processes. As seen the instrumentation process reached 4500 iteration before it ended prematurely, resulting in an average of 330,5 seconds per iteration. For the process only performing fuzzing, and no instrumentation, a total number of 19769 iterations were performed yielding an average value of 50,8 seconds per iteration. Based on these two average values, the pure fuzzing process is roughly 6,5 times faster than performing instrumentation with the given PIN tool. As these two numbers are very different, and I cannot with certainty say which is the most correct one (although the maximum number of iterations typically yields a more representative result), some additional time testing was performed, which will be presented shortly.

Normal fuzzing performs each iteration on average a lot faster than the smart-fuzz tool which performs several additional time consuming processes. The major contributor is the instrumentation process where the target program runs with additional code for each BBL it executes. As the number of BBL increases the additional time needed increases respectively.

To be sure to not add two similar traces to the backlog, each new trace for the mutated files doesn't only have to be compared to the trace of the current sample, but also with all traces added to the backlog throughout the execution of the smart-fuzz tool. These two comparison processes also consumes a lot of time as each element in the trace to investigate must be compared to each element of the other trace in question. All together these to contributing factors are responsible for most of the additional time used.

What we can identify from the results of the smart-fuzz process with the two samples that initially were the most different of all the samples of the minset is that in both cases the smart-fuzz tool was able to utilize feedback from instrumentation and find a new sample that was better based on the given criteria. The new samples for the two different instances of running the fuzzing tool yielded an increase of 1,4 and 1,9 in code coverage. When looking at the number of BBLs executed it is important to note that the address of each BBL entry is compared to all entries of the trace of the other sample and only differences are counted. This means that it is impossible to directly see from the number of BBLs in a sample's trace if it is more or less different from another sample when considering the threshold value.

Another key observation that should be highlighted based on the same results are that in both cases there is no improvement in the context of the number of new good samples found when increasing the number of iterations from 300 to 1500, a 5 times increase. This can be seen as only one good sample is discovered for both of the processes and they contain the same values for CC and for the size of the trace.

What is curious to see is that the new samples found for the two different original base samples yield almost the exact same CC and size of the trace. Whether this is a coincidence or if there are some other explanations is uncertain. Another interesting aspect to take note of is that the last of the three initial samples, the biggest one with the greatest CC, yielded 4 new samples in comparison to only one new sample from the two other initial samples. It could seem like bigger samples have a greater potential of producing samples with noteworthy distinctions.

For the additional research conducted to get more data, the first part when comparing normal fuzzing with smart-fuzzing for the smallest sample did not yield much interesting results. We can see that neither normal fuzzing nor the smart-fuzzing tool were able to produce any sample with greater code coverage than that of the initial sample, 83,2 %. This could be due to the properties of the sample used. A small sample will theoretically have fewer fields and data to mutate, and thus less chances of making major changes in the mutated sample. This should be investigated further; does the size of the samples used influence the time needed by instrumentation? We can however see that for the small sample used, the normal fuzzing is 9,2 times faster than the smart-fuzz tool (calculated based on the total run time for both techniques). This value is not too far off from the factor

calculated based on the time elapsed for 19769 iterations with normal fuzzing and 4500 iterations for both fuzzing and instrumentation; 6. These result seem to indicate that both there are aspects that may affect the process time wise (such as sample time and system load) and that when adding instrumentation the total time increases with a factor of between 6 and 9 compared to solely fuzzing.

It should also be noted that the efficiency of the smart-fuzz tool is somewhat dependent of the strength of Peach in the sense that it is able to perform good mutations, which in turn is dependent on having a good Pit file for the target file format.

The smart-fuzzer is unarguably slower than normal fuzzing, which should be expected due to the number of additional operations that are performed. The two main contributors seem to be the instrumentation process and the process of comparing traces. However, the number of traces to compare for each iteration should remain rather low, but instrumentation of every single execution unit in the target program will occur each iteration. To be able to optimize the instrumentation process should therefore also greatly increase the speed of the smart-fuzz tool. From the results of the additional minor research conducted for the third research task, it can be seen that the smart way of instrumenting is almost three (3,2) times faster than the normal way of instrumenting. There are some fluctuations in the results, but this is realistic as the different samples used can trigger different and new parts of the target program and there may also be some minor external operations running affecting the available resources when performing the time measurements. As can be seen for the smart instrumentation technique, there are still some time required for each iteration, which is due to the fact that the target program will have to start up through PIN and perform its action even tough little additional code is inserted into the target program. Also, the first iteration for both techniques require the same amount of time, which is due to the fact that the smart instrumentation technique only instruments new execution units, and for the first iteration, each execution unit will be evaluated as a new execution unit for the given run. One important note when using the smarter way of instrumenting, as briefly described with the results presented, is to be aware of that the way of calculating the code coverage and producing the trace also has to be adapted to the instrumentation process not running the inserted code at each execution unit, as this was the original way of performing such metrics.

Section III

8 Conclusion and further work

8.1 Code Coverage

- *How far from a trace does a bug actually occur?*

This section deals with the results from the first research task; Code Coverage. A description of the term research task has been given in Section 4.3.1, Nature of the thesis.

An original trace (for a sample) is simply the first trace of the execution of a target program with one specific (non-mutated) sample. The trace for a program with a sample before mutation is referred to as an original trace, while the trace for the same program with a mutated version of the same sample is no longer an original trace.

The hypothesis of this research task is that having higher code coverage of the target program will result in a greater number of faults detected during fuzzing due to the fact that most crashes that occur are typically situated close to the original trace of the target program. In other words, what we wanted to investigate was whether crashes typically were close to the original trace, or not.

To investigate this research task's hypothesis, a low number of hops (function calls) between the crashed function and the original trace for the crashes investigated for the two target programs on average is required. As seen, in Section 7.1.2**Error! Reference source not found.**, Results, for all crashes for both target programs the distance (number of hops) were calculated to 0, which is as low as it gets. This means that all crashed functions are indeed *on* the original trace for all investigated crashes. Based on these results we can conclude that crashes are indeed very close to the original trace, and thus the hypothesis is seemingly correct based on these results. This in turn substantiates the importance of high code coverage when fuzzing programs. *By having a high level of code coverage of the target program when fuzzing there is a much greater chance of finding exploitable instruction or instructions that causes crashes, due to such locations typically being very close to the original trace.*

The fact that all values for the distance are zero might seem somewhat suspect as one of the strength of fuzzing is to reach special branches of the program not normally taken during normal execution due to unanticipated and mutated values, which should in fact result in reaching functions not being a part of the original trace. However in small programs and programs where insufficient means of validating data before using it in operations such as branch selection has been used, it is not unlikely that instructions causing crashes are also

located in the original trace. As both the target programs used are of early versions and having a focus on being lightweight it is reasonable to believe the results to be accurate and valid. There is also a possibility that there is an error in the secondhand code that has been developed for this project that is the cause of the results mentioned.

In addition to discovering both a method for how far from a trace a crash actually lies, there was one other aspects that was looked into in this research task. When working with traces and code coverage I found it interesting to also create a technique for discovering where a trace with a normal sample differentiated itself from a trace with a mutated version of that same sample. Some interesting results were found, but most interesting was to see that the created technique works, yielding the last common address for a split as well as the next address for both the original trace and the mutated trace.

From the results we have also seen that there are some addresses that occur repeatedly indicating functions that could be of interest. This technique can also be used to see what mutation techniques that produce different code paths, which could result in different entries in the tables. The information in the results can also be used when debugging an application if interested in closer inspecting the various functions that are listed.

The differences between the different samples are interesting in the way that they support the fact that various samples do affect the program trace, or code path, during execution. It is also possible to identify some locations that are common for several of the traces for the samples used which could be of interest.

8.2 Pre-fuzz

- *Investigate the importance of templates and their impact on fuzzing results. Adding features to the template and investigate differences in results, trace and code coverage. Learn what parts of the data model that are interesting. Determine to produce samples by creating them from a valid program or generating from a data model.*

This section deals with the results from the second research task; Pre-fuzz. A description of the term research task has been given in Section 4.3.1, Nature of the thesis.

Before entering this section I will recap that I differentiate between creating samples, denoting to make a sample by saving the content of an editor to a file, and generating samples which denotes to make peach produce samples from a data model/ input data file.

This research task is aiming at discovering how some of the properties of samples affect the fuzzing process and the fuzzing results. Several various properties have been looked into. The first property that was investigated was whether it would be efficient to add more features to the samples, and to see what features that gave the biggest increase in code coverage (as we have seen from the first research task that having a good code coverage is important for good fuzzing results). Adding feature yields a better code coverage than a blank document, and thus would probably be a better sample for fuzzing. What feature to add depends on several variables, and will be further discussed. What is interesting to see is that adding one feature does not increase the total code coverage a lot, but there are some differences between the best and the worst feature to add (in the context of increased code coverage). Furthermore it is clear to see that a combination of feature yields more than any of the selected features alone, making it reasonable to adapt this practice when producing samples, potentially guided by the list of features that increased code coverage the most.

When considering the results for the code coverage for each unique feature it is clear to see that what features to focus on depends strongly on the target program as they may have different functionalities and aims (xPDF focusing on text inside the document, but SumatraPDF is rendering the document). What feature that is the most important to focus on will therefor vary depending on both the file type and the target program. *Thus there is no clear answer for what feature that is the best to add, but having a complex document consisting of a combination of features while also spanning over several pages will greatly improve the code coverage of the sample. By using the presented technique one may be able uncover the most appropriate feature for any given file format and target program.*

Another interesting aspects of the samples that was investigated in this research task was the production technique of the samples, is it best to utilize generated samples or to create the samples from a valid program? To be able to give an answer to this question most of the operations in this research task have been conducted with both a set of created and generated samples, where the same features are present in both set of samples. When looking at the process of discovering the difference in code coverage for the various samples with different features in comparison to a blank document, with SumatraPDF as the target program, the created and generated samples yielded pretty much the same results. However for some of the samples the generated sample did get 0,1 percentage point higher score. When performing the same operation on xPDF the results were identical for generated and created samples. In addition to this data, the features were also combined and a created and a generated counterpart sample were produced and the code coverage was measured. For SumatraPDF the results were equal, but for xPDF the created file got 0,1 percentage point higher score than the generated file.

From this we can conclude that in most scenarios, when the created and the generated file have to contain the same feature and content (to be able to compare them in a realistic way) there are very small variations. In many scenarios they are equal good, in some other

generated samples is somewhat better, but then again the created samples may be better in some other scenarios. *Thus we can say that based on the steps taken in this thesis and the appurtenant data; there is not any reason to choose one technique over the other. The technique should be picked based on other criteria such as accessibility of samples, required time to get a sufficient number of samples and knowledge of the data format (for generation).*

The last area of interest in this research task is to see if it is possible to discover key areas of the samples (file format) that are more important to focus mutations on. To find a good answer for this problem all produced samples were compared to a blank document to see what segments of the files that were different. If a given segment occurred many times for the total number of samples tested, this segment would indicate an area of the file format that is likely to always be executed by the target program. Thus this section is likely to be thoroughly tested for bugs. Mutation should therefore not be focused on this area, but rather areas that seldom occur, to produce different kind of samples with the main differences in sections that are hard to reach. From the calculations done and the results it is clear to see that there are indeed such locations. One can therefor conclude that it is indeed possible to discover areas that are more interesting than others. The technique presented can be used on all kind of file formats and will produce different results for each type of file format.

We can also see from the minset operation that more samples do not necessarily give higher total code coverage.

Lastly, as one of the assumptions I'm working with is that there should be no prior knowledge of the target program, some might argue that investigating the file format and using this information to make improvements is in fact to have prior knowledge before fuzzing. However I would claim that when performing the operations described I'm operating at a "feature level" and not at an application level and thus do not violate the assumption in question. Furthermore, the information in question is also extracted from the available samples, and is not acquired by having prior specific knowledge of the target program.

8.3 Fuzzing

- *Use feedback from instrumentation to generate new fuzz cases.*

This section deals with the results from the third and most important research task; Fuzzing. A description of the term research task has been given in Section 4.3.1, Nature of the thesis.

To recap, an original trace (for a sample) is simply the first trace of the execution of a target program with one specific (non-mutated) sample. The trace for a program with a sample before mutation is referred to as an original trace, while the trace for the same program with a mutated version of the same sample is no longer an original trace.

As we can see from the time measurement results, there is a pronounced difference when looking at the ratio of the time elapsed for fuzzing with instrumentation and just fuzzing after 1000 iterations and the average time; 45,7 and 6,5 respectively. In my eyes this simply strengthens my claim that more iterations will have more representative results when compared to less iteration. The important point to be made from these results is that there is a substantial increase in the time require when instrumenting each fuzz case.

Due to the fact that the extra time required is so substantial, it might in fact be more profitable to simply do additional normal fuzzing iterations instead of instrument each fuzz case, as one on average could complete 6 additional fuzzing runs in the extra time required for instrumentation. To see if this indeed is the fact, we also need to see whether the smart-fuzz tool actually is able to find new and better samples.

As seen the smart-fuzz tool was able to discover at least one new sample that was considered better based on the given criteria for all of the three initial supplied samples, and in certain occasions more than one. This is proof enough that it is possible to make smarter decisions in dumb fuzzing without having any prior knowledge of the target program.

Even though there are no results of any potential increase in the number of unique crashes uncovered by fuzzing with these new samples, each and all of the new found samples have a greater code coverage than their source sample and as seen, greater code coverage is of importance when attempting to find as many bugs as possible. Thus we can conclude that even without actual empiric results to refer to, one would benefit from using feedback from instrumentation for having a greater bug discovery rate when fuzzing.

There is however certain aspects of the process that needs to be improved. The most important one is to incorporate crash logging of the fuzzing process in the smart-fuzz tool so that one directly can discover bugs and flaws in the target program while utilizing the smart-fuzz tool. More information on why this has not been done and the obstacles to counter can be found in section 8.5 Further work. By having information of the unique crash yield rate would greatly strengthen the basis used to conclude from regarding the efficiency and importance of the smart-fuzz tool. The way the tool is constructed now calls for the need to start another clean fuzzing process with the individual new good samples uncovered by the smart-fuzz tool.

Furthermore it is somewhat suspect that the one good sample found for the two initial samples with the most different trace has almost the same code coverage and also almost identical size of the trace. As mentioned this could be a coincidence, but it still seems somewhat suspect. As to any theory of why this is, I have not.

Finally, it is unfortunate that concrete data for the four new good samples discovered from the last instance of the smart-fuzz tool with the biggest sample with the greatest code coverage is unavailable. These results could prove highly interesting, and should certainly be produced if time allows it. However the main aim of this research task was to create a fuzzer tool making use of feedback from each fuzz case to improve the fuzzing process and increase efficiency when having no prior knowledge of the target program, as well as show results supporting the fact that the produced tool works as anticipated with real life applications and samples. I believe that the concepts, principals, techniques, data and results presented in conjunction with this research task shows that the aim has indeed been met, even though there still are room for improvements.

The very last issue to discuss, which I have briefly mentioned, is whether it is time efficient to instrument each fuzz case or not. As of now, when a new clean fuzzing process with each of the found new good samples will have to be done to log crashes due to the problem of attaching winDBG to several applications at the same time, I would say that it is not time efficient. Also due to the lack of actual results on whether these new samples increase the number of unique crashes or not, it is even harder to make a clear decision. So as stated as of now I do not believe it is more time efficient, but by overcoming one last obstacle I believe it will. Also as seen, the instrumentation of each fuzz case adds a substantial amount of time to each iteration and I therefor believe that one might uncover more unique crashes within the same time window used to perform a full run a the smart-fuzz tool.

Additionally when looking at the results from the additional research conducted, when running a small sample with the smart-fuzz tool and with normal fuzzing, we can see that the smart-fuzz tool does not yield any benefit over the normal fuzzing. As mentioned this could be due to the small sample used, and it could seem as the smart-fuzz tool is not useful with smaller samples. During this research 1000 iterations were performed and the difference in the time required could be described as a factor of 9,2. This is a major deviation from the difference factor found when uncovering the baselines for the time requirements at the beginning of this master thesis when running a huge number of iterations which was calculated to be 45,7. On the other hand it resembles the factor found for 1000 iterations; 6,5. The differences in the time recorded can be both explained by varying available resources at the time of execution, but it is also possible that different kind of samples have an impact. The latter should be further investigated. However the 45,7 factor does still deviate too much from the other two factors produced, that it is reasonable to believe that it is only caused by the two aspects mentioned. It could also be argued that running several thousand iterations yield a more correct average time than only running 1000 iterations, but 1000 iterations should still yield a sufficiently accurate result.

Lastly it should be mentioned that an attempt to optimize the instrumentation process has been presented which as seen greatly reduces the time needed for the instrumentation. As the smart-fuzz tool utilizes instrumentation for each iteration, the smarted way of

performing instrumentation will also increase the overall speed of the smart-fuzz tool. On the ten samples used during testing, the smart way of performing instrumentation is more than three times faster than the normal way of performing the instrumentation, which is currently used in the smart-fuzzing tool. For further work with this topic, the smart instrumentation technique should be adapted by the smart-fuzz tool.

8.4 Answering the research question

In this section answers to the research questions will be provided based on the conclusions from the various research tasks representing the conducted research and appurtenant results. The research question answers will serve as the final overall results of this thesis. Firstly the first research question is recapped:

RQ1: Does the templates used during file format fuzzing impact the fuzzing process and results significantly, and if so, how?

First off we can look at the results from the first research task, which investigated whether a crash normally was close to the original trace or not. What was concluded from this research task was that a crash typically was located very close to the original trace. This in turn means that there is a great need of having templates with good code coverage to be able to cover as much of the programs code base as possible, and thus have a higher chance of triggering a bug when using the template for mutations during the fuzzing process. In other words we can say that the templates used, impacts the chance of finding bugs, and thus also the efficiency of the fuzzing process. The results of the first research task does however not state anything about how the template should be nor how changing it may affect it, simply that having a good template yielding a high level of code coverage is important.

The results of the second research task are focused on the content of the templates and how changing certain aspects can change the efficiency. The first thing to be seen is the difference in code coverage of the target program when one by one feature is tested. As have been discussed, there aren't that much of a difference between the feature that increases the code coverage the most and the feature that increases the code coverage the least, but it is certainly noticeable. Furthermore when combining all features the increase in code coverage become even more prominent as there was an increase of 44 % for the result from SumatraPDF and as much as 70 % for xPDF. The total code coverage was however quite small; 15,6 % and 11,9 %, respectively. Even though the total code coverage wasn't too great, it is clear to see that the content of a sample, in the context of the various features covered for this example has a noticeable effect on the code coverage, which, as seen, should affect the fuzzer's chances of finding an increased number of bugs.

The same research task also investigated if there was a noticeable impact on the results of the fuzzing process when either using samples that had been created or when using samples that had been generated based on a data model and cracked data from the created samples. The results from the research done at this particular area show that when using the created samples to crack data from (to be sure to have the same functionality, or when possessing an insufficient data model), the resulting generated samples do not differ much from the created counterparts. The order of the features having the most impact on the code coverage have changed around somewhat and there are minor differences in the total code

coverage, but nothing noteworthy. When talking about the number of produced crashes for a set number of iterations, the generated samples uncovered more crashes during the fuzzing process than the created samples when using xPDF as the target program, but the opposite is true when fuzzing SumatraPDF. This could have been a coincidence, or this could be a trend describing the fact that some target programs are better to be fuzzed with generated samples, while others are more suited for created samples.

The overall answer to the question is that, yes, certain aspects of the templates do in fact impact the fuzzing process, but certain other aspects do not. Which of the aspects that have an impact, have just been discussed.

The second research question is concerned with the idea of having an automatic way of learning from the ongoing fuzzing process to actively improve it. To recap the second research question:

RQ2 (Main RQ): Is it possible to improve the effectiveness of a file format fuzzer by focusing on the template used based on feedback from instrumentation.

The short answer to the process is “Yes, it is possible”. A tool using feedback from instrumentation to make smarter decisions and which will improve fuzzing has been created and which has been shown to give the desired results, has been presented. The tool also requires no prior knowledge of the target program.

Throughout the last research task of this paper, concepts and ideas for an automatic fuzzing tool making smart decisions have been presented. A thorough explanation of the construction and the techniques utilized in the tool has also been shown. Furthermore based on the results presented it has also been proven that the tool is able to achieve its desired purpose.

The tool has room for improvements, whereas a specific one is believed to dramatically increase the usefulness of the created tool. The improvement in question is the ability to incorporate crash logging directly into the tool excluding the need for an addition fuzzing process with the better samples produced by the smart-fuzz tool. Additionally by adapting smart instrumentation the time required should also be greatly reduced, which in turn also increases the efficiency of the tool. Due to some of the current drawbacks of the tool I was unable to get any results using the metric “unique crashes produced” when utilizing the tool, which is regretful as such results would definitively be the best ones to assess the efficiency and usefulness of the tool.

However with this in mind, it is clear to see that the tool is able to identify new and better samples, and based on other research and trends it is reasonable to believe that better samples yield more unique crashes, which in the end is what we are aiming for.

As discussed, whether it is more time-efficient to use the smart-fuzz tool or simply using normal fuzzing has not been given a clear answer, but techniques to do so will be presented in the Further work section. Personally I do believe that when having addressed the one major drawback of the tool for its current version, it will indeed be more efficient than normal fuzzing with regards to how many unique crashes that are uncovered during a given time interval.

Even though the tool has room for improvements, the goal of the research question has been reached and results have been produced.

8.5 Further work

After working with the topics of this thesis I feel that I have accomplished the goals and aims defined. Among other aspects, I have also personally gotten a broader understanding of related concepts and techniques. Even though I am content with the final thesis result and the paper, there are certain aspects that I would have liked to dedicate more time to, if available. These aspects are for instance peculiar findings during the research process that I do not know the cause of, but can also include certain areas where I would like to further research the behavior and effect of certain techniques and concepts. This section of the paper will briefly discuss these aspects, which should be considered to be suggestions for further work on the topic.

When investigating to see if there are any major differences between creating and generating samples, I identified a potential tendency concerning the outcome of the performed research which should be looked into further. What I identified was that when using the created and the generated samples to fuzz the first target program, SumatraPDF, more unique crashes were discovered for the fuzzing runs with the created samples. The code coverage was very much the same for both the created and the generated samples when used with SumatraPDF. However, when running both kinds of samples with the second target program, xPDF, the generated samples were the ones that produced the most unique crashes. From these results it seems like the created samples are to be preferred when attempting to uncover as many unique crashes as possible with one program, while the generated samples are more preferable when targeting another program. What I would suggest for future work is to further investigate this trend to identify if this observation is purely random for the two target programs that I have tested or if there is indeed something to the presented observation. This can easily be done by using the same set of samples; both created and generated, while fuzzing more target applications. It could be of great interest to get a broader base of result to conclude from on this topic.

During some of the fuzzing runs extremely large samples were produced. These files would at some times consume all the remaining free space of the storage medium, which could be as much as 450 GB. The initial sample files used for the mutations had a size ranging from 1 KB to 181 MB. When the large files were produced Peach would typically crash as it wanted to create even bigger files but was unable to due to the lack of extra free space. Depending on how Peach was used and in what context, these crashes could result in the termination of a bigger wrap-around process. I tried to enforce several limits on how Peach should perform the mutations by tweaking variables in the Pit files used, but was unable to both uncover the cause of these huge file and thus finding a solution. I would like to understand why and how these huge files were created, and it could indeed by the scope of some future work on the topic. Solving this conundrum could also help others in future work were the same behavior could be observed.

When testing the functionality of the smart-fuzz tool I focused on whether it was able to uncover and identify new and better samples compared to the one currently used based on the feedback from instrumentation. As seen from the results, this was indeed the case. However I never investigated the potential benefits with regards to the number of crashes or bugs discovered when using these new samples, partially due to insufficient time. This however would provide valuable information on the effectiveness of the fuzzer. It could also be used to correctly assess whether it is most valuable to perform fuzzing with instrumentation to make smarter decisions or simple more iterations with normal fuzzing. One way of performing this process could be to take the new samples from the smart-fuzz process, together with the initial sample, and run them through a normal fuzzing process with the same values for the range and seed parameters and directly compare the number of unique crashes discovered. The most optimal solution would however be to incorporate the fuzzing directly into the smart-fuzz tool, which initially also was the idea. However there are some problems with the debugger when attempting this approach which has to be overcome. When the WinDbg was attach to the fuzzing process, an error was discovered when trying to attach it to the PIN process started from the fuzzing engine, as it all was already attached to the peach process. Both peach and PIN utilize the debugger. Peach use it to identify and log crashes, while PIN uses it to enable it to perform instrumentation.

As seen when analyzing the results from the smart-fuzzing tool, one of the major contributors for the big difference in the time required to run the tool compared to normal fuzzing was caused by the instrumentation process. It could be interesting to see if there are major differences on the time the instrumentation requires when running with one kind of samples or another. More specifically, will for instance a very small sample complete the instrumentation process much faster than a much bigger sample, or is the properties of the sample used close to irrelevant for the instrumentation process? On this topic, but not concerning the type of samples used, a suggestion for improving the time required for instrumentation has been presented. The smarter, and faster, way of performing instrumentation has briefly been presented at the end of Section 7.3.2, Results, and

discussed in the appurtenant sections. For further work with this topic, the smart instrumentation technique should be adapted by the smart-fuzz tool. When doing so there should be given extra care to adapting the way of producing code coverage and trace information as the currently used technique will be affected by the changes done in the smart instrumentation technique presented.

Section IV

9 Table of figures:

10 Table of tables:

11 Index

12 Bibliography

Bibliography

1. Wang, T., et al. *TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection*. in *Security and Privacy (SP), 2010 IEEE Symposium on*. 2010. IEEE.
2. McGraw, G., *Software security.* Security & Privacy, IEEE, 2004. **2**(2): p. 80-83.
3. Takanen, A., *Fuzzing: the Past, the Present and the Future.* Codenomicon Ltd, 2009.
4. Takanen, A., et al., *Fuzzing for software security testing and quality assurance*, in *Artech House information security and privacy series*2008, Artech House,: Boston ; London.
5. Iozzo, V., *0-knowledge fuzzing.* 2010.
6. Charlie Miller, Z.N.J.P., *Analysis of Mutation and Generation-Based Fuzzing.* 2007: p. 7.
7. Godefroid, P., M.Y. Levin, and D. Molnar, *Sage: Whitebox fuzzing for security testing.* Queue, 2012. **10**(1): p. 20.
8. Ostrand, T., *White-Box Testing*, in *Encyclopedia of Software Engineering*. 2002, John Wiley & Sons, Inc.
9. *Fuzz Testing of Application Reliability*. 2008 Sun May 25 20:44:07 CDT 2008 [cited 2013; Available from: http://www.cs.wisc.edu/~bart/fuzz/.
10. Amini, P., A. Greene, and M. Sutton, *Fuzzing: Brute Force Vulnerability Discovery*, 2007, Addison Wesley Professional.
11. OUSPG. *PROTOS Security Testing of Protocol Implementations.*; Available from: www.ee.oulu.fi/research/ouspg/protos/.
12. Microsoft. *Microsoft Security Bulletin MS04-028*. 2004 14/12 - 2004 [cited 2009 02/09]; Available from: http://technet.microsoft.com/en-us/security/bulletin/ms04-028.mspx.
13. Howard, M. and S. Lipner, *The security development lifecycle*. 2009: Microsoft Press.
14. RUDERMAN, J., *Introducing jsfunfuzz. Blog Entry*, 2007.
15. Holler, C., K. Herzig, and A. Zeller. *Fuzzing with code fragments*. in *Proceedings of the 21st USENIX conference on Security symposium*. 2012. USENIX Association.
16. Incorporated, A.S. *Adobe Secure Product Lifecycle*. 2010 [cited 2013 05/05]; Available from: http://www.adobe.com/security/pdfs/privacysecurity_ds.pdf.
17. Aitel, D., *An introduction to SPIKE, the fuzzer creation kit.* immunity inc. white paper, 2004.
18. Sutton, M. and A. Greene. *The art of file format fuzzing*. in *Blackhat USA Conference*. 2005.
19. *Peach Fuzzing Platform*. 2013 02/07 - 2013 [cited 2013 02/09]; Available from: http://peachfuzzer.com/.
20. DeMott, J., R. Enbody, and W.F. Punch, *Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing.* BlackHat and Defcon, 2007.
21. Microsoft. *Introduction to Instrumentation and Tracing*. 2013 [cited 2013 11/09]; Available from: http://msdn.microsoft.com/en-us/library/aa983649(v=vs.71).aspx.
22. OC-Systems, I. *Experience the Power of Software Instrumentation*. 1999-2013 [cited 2013 12/09]; Available from: http://www.ocsystems.com/softwareinstrumentation.html.
23. Intel. *BBL: Single entrance, single exit sequence of instructions*. 2013 [cited 2013 12\09]; Available from: http://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__BBL__BASIC__API.html.
24. Intel. *Pin - A Dynamic Binary Instrumentation Tool*. 2012 [cited 2013 19 - 02]; Available from: http://pintool.org/.
25. GCC-team. *GCC, the GNU Compiler Collection*. 2013 [cited 2013 11\09]; Available from: http://gcc.gnu.org/.
26. Jones, M.T. *Visualize function calls with Graphviz*. 2005 [cited 2013 11/09]; Available from: www.ibm.com/developerworks/library/l-graphvis/.
27. Kowalczyk, K. *Sumatra PDF*. 2013 [cited 2013 19\09]; Available from: http://blog.kowalczyk.info/software/sumatrapdf/free-pdf-reader.html.

28. Yan, W., Z. Zhang, and N. Ansari, *Revealing packed malware.* Security & Privacy, IEEE, 2008. **6**(5): p. 65-69.
29. Krohn-Hansen, H., *Program Crash Analysis: Evaluation and Application of Current Methods.* 2012.
30. fooLabs. *Xpdf*. [cited 2013 15/11]; Available from: http://www.foolabs.com/xpdf.
31. *Xpdf for Windows*. [cited 2013; Available from: http://gnuwin32.sourceforge.net/packages/xpdf.htm.
32. Déjà-vu_Security. *Déjà vu Security - Secure Development & Security Advisory Services*. 2013 [cited 2013 18/09]; Available from: http://www.dejavusecurity.com/.
33. University_of_Virginia. *Pin - A Dynamic Binary Instrumentation Tool*. 2012 [cited 2013 19/09]; Available from: http://www.cs.virginia.edu/kim/publicity/pin/.
34. Microsoft. *Visual Studio*. 2013 [cited 2013 19/09]; Available from: http://www.microsoft.com/visualstudio/eng/visual-studio-2013.
35. Zynamics. *zynamics BinNavi*. 2012 [cited 2013 13/10]; Available from: http://www.zynamics.com/binnavi.html.
36. Hex-RAys. *IDA: About*. 2012 25/07 - 2012 [cited 2013 13/10]; Available from: https://www.hex-rays.com/products/ida/index.shtml.
37. Hex-RAys. *Executive Summary: IDA Pro – at the cornerstone of IT security*. 2009 25/07 - 2012 [cited 2013 24/10]; Available from: https://hex-rays.com/products/ida/ida-executive.pdf.
38. Eagle, C., *The IDA pro book: the unofficial guide to the world's most popular disassembler*. 2008: No Starch Press.
39. Microsoft. *Windows Debugging*. 2013 10/12/2013 [cited 2013 25/10]; Available from: http://msdn.microsoft.com/library/windows/hardware/ff551063(v=vs.85).aspx.
40. AndyRenk. *!exploitable Crash Analyzer - MSEC Debugger Extensions*. 2009 2013 May 2 2:17 AM [cited 2013 25/10]; version 10:[Available from: http://msecdbg.codeplex.com/.
41. SweetScape. *010 Editor - Edit Anything*. 2002 - 2013 [cited 2013 27/11]; Available from: http://www.sweetscape.com/010editor/.
42. Mark Russinovich, D.A.S., Alex Ionescu, *Windows Internals, Fifth Edition*. Microsoft Press.
43. SweetScape. *010 Editor - Comparing Files*. 2002-2013 [cited 2013 21/11]; Available from: http://www.sweetscape.com/010editor/manual/Compare.htm.

13 Appendix

The appendix will be available on CD bundled with the master thesis and will contain:

- Peach 2 Pit files
- Peach 3 Pit files
- Pin Tools
- All code, organized in one folder for each of the research tasks