UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Gray-box Network Fuzzing using Genetic Algorithms and Code Coverage

Fokke Heikamp
M.Sc. Thesis
October 2017

Supervisors:
prof. dr. J.C. van de Pol
dr. A. Peter
ing. R. van Hees

Formal Methods and Tools Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

THALES

Summary

There are many recent news articles on the topic of cyber-security. This includes articles about companies being hacked, theft of digital information and the spread of ransomware. A big component in all these incidents is software. Defects or vulnerabilities in software might allow malicious entities to gain access to computer systems. These vulnerabilities are usually easy to remove if they are known. Finding them is however difficult since software is large, complex and connected.

Fuzzing is a method to automatically find vulnerabilities for a given target. The basic idea of fuzzing is to generate lots of data, send it to the target and see if the application handled it correctly. Random fuzzing has shown to be effective however it suffers from a serious drawback. It generally has a poor code coverage because of its random nature, which in turn means that a relatively large part of the target remains untested.

To solve this problem a new fuzzer, called MyFuzzer, was developed. MyFuzzer contains a genetic algorithm which tries to maximize code coverage. Each input gets scored on eleven metrics. Most of these metrics quantify how much new code has been executed. The two main metrics used in MyFuzzer are the number of new basic blocks found by an input and the number of new transitions between basic blocks found by an input. A basic block is a list of instructions without jumps. The genetic algorithm calculates a score for an input based on its metric values. Inputs which find a lot of new code get a higher score and have more probability of being reused. MyFuzzer has two main design restrictions. First of all the set of targets is limited to network applications and secondly the target source-code cannot be used. This means that code coverage has to determined on instruction level.

After the development of MyFuzzer research is done by executing several instances of MyFuzzer on a set of targets. The first goal of the research was to show that a genetic algorithm improved fuzzing both in terms of finding vulnerabilities and code coverage. The second goal of the research was to determine which metric or combination of metrics best describe the fitness of an input.

The results for this project suggest that a genetic algorithm does improve fuzzing because MyFuzzer found vulnerabilities more frequently and obtained a better code coverage than fuzzers without a genetic algorithm. The results also indicate that the number of new basic block found by an input best describes the fitness of an input because focusing on new basic blocks delivered the best code coverage results.

Contents

Chapter 1

Introduction

Software is everywhere these days. It is used for email, playing games, writing reports, bookkeeping, controlling hardware, storing files and more. It is used in consumer products as well as in embedded systems. Governments, companies and consumers are all heavily reliant on software. Because of this reliance it is important that the software does what it is supposed to do and is protected against malicious use. This is easier said than done. An inspection of the national vulnerability database [24] or the exploit database [30] shows that software is rarely secure. Software testing can be used to identify software failures, however finding all software failures is generally an undecidable problem [26]. Another limitation of software testing is that it can only show the existence of a software failure, it cannot show the absence of software failures [26]. Because of these theoretical limitations the main goal of software testing is to minimize software failures by identifying as much failures as possible.

The focus for this project is on automatically finding security failures for a given target. Security failures are defects in a piece of software which can be misused by a malicious entity. There are several automated security testing techniques available, the main methods being symbolic execution, taint analysis, model based security testing and fuzzing. The last method was chosen for this project. The most basic form of fuzzing works by sending random inputs to the target. More complex fuzzers use additional knowledge about the target to generate better test-cases. For this project a new fuzzer called MyFuzzer was created. MyFuzzer extends the basic fuzzing functionality with support for genetic algorithms which try to optimize runtime code coverage. Each test-case gets a score which indicates the importance of that test-case. The scores are based on runtime code coverage metrics. The code coverage metrics quantify how much new code a test-case has found in the target. A test-case which finds a lot of new code is more likely to be used again, whilst a test-case which does find little new code is likely to be removed. Some important restrictions were placed on the fuzzer. First of all the source-code of the target is not available. This means that the code coverage has to be obtained on assembly level. The reason for this is that source-code is not available in closed-source products. Secondly the set of targets for MyFuzzer is restricted to network applications.

The project was carried out for Thales Hengelo. Thales Hengelo is known for developing products like the SMART-L, APAR and the GOALKEEPER. Thales Netherlands is also known for their work in public transport and cyber-security. MyFuzzer was developed to be tested on the software they develop in order to increase the confidence that their software is secure. Software security is important for the products Thales develops because their software is used for critical applications.

1.1 Motivation

Software testing is expensive, difficult and time consuming [26]. The main reason for this is that the input space for a given SUT is very large. A tester has to take a representative sample of inputs/test-cases from all possible inputs. MyFuzzer solves this problem by using a genetic algorithm. Test-cases which find a lot of new code in the target have more chance of being used again in a mutated form. Test-cases which do not find a lot of new code have less chance of being selected and are likely removed. The benefit of selection is that less promising inputs don't have to be evaluated anymore, which saves resources. The aim of this approach is to guide MyFuzzer into the right direction whilst keeping its power of random mutation.

This project has two goals:

1. To investigate the practicality of gray-box network fuzzing using a genetic algorithm optimized for maximizing runtime code coverage.

2. To do research about optimal parameter settings for the genetic algorithm.

The first goal is about creating a practical fuzzer. This means that MyFuzzer should not be reliant on hard to obtain information. It should also not require time consuming pre-execution steps in order to work. MyFuzzer should be able to run after a small setup and generate results which are easy to understand. The practicality of MyFuzzer can be measured by comparing it to Sulley. Sulley is perfect as a baseline because MyFuzzer is based on Sulley.

The second goal is about the parameters of the genetic algorithm and their influence on the performance of MyFuzzer. The main question for this goal is ”which parameter settings deliver the best results”. Results can be evaluated on the number of vulnerabilities found, code coverage and the time it took to find vulnerabilities.

1.2 Research Questions

The goals mentioned in 1.1 can be used to construct the following research questions:

- Can the accuracy of a gray-box fuzzer be increased by using a genetic algorithm optimizing for runtime code coverage.

    1. How does MyFuzzer compare to existing fuzzers in terms of finding vulnerabilities and code coverage.
    2. How does the choice of metrics for the fitness function influence the accuracy of MyFuzzer.
    3. How does the choice of fuzzing strategy influence MyFuzzer in terms of finding vulnerabilities and code coverage.

Research question 1 can be tied to the first goal explained in 1.1. The other two research questions, 2 and 3, are related to the second goal explained in 1.1.

Terms like fitness function, gray-box fuzzing and fuzzing strategies will be explained in more detail in chapters 2 and 3.

1.3 Rationale

The rationale behind the solution presented in this report is based on several assumptions. The first assumption is given in assumption 1. It is easy to see that this assumption holds for most instances. For example take a piece of code which has n possible test-cases, then each unique test-case which does not find a vulnerability decreases the probability that the piece of code contains a vulnerability. It is still an assumption because it does not state that each test-case is unique.

Assumption 1 *The more times a piece of code has been tested the less likely it is that it contains vulnerabilities.*

The second assumption 2 states that the focus should also be on the least tested component. This assumption is based on the first assumption. Intuitively assumption 2 makes sense. If part a has been tested extensively whilst part b has only been tested once it makes sense to test part b more often.

Assumption 2 *It is a good practice to focus on testing parts of the SUT which are not well tested.*

Of-course following these assumptions does not give guarantees about the test quality. First of all a vulnerability is only observed when the target receives data which triggers the vulnerability. Consider a buffer overflow vulnerability where there is room for 100 bytes. The vulnerability will only be triggered if it receives an input larger than 100 bytes. Secondly the location or state of the target where the vulnerability can be triggered has to be found. If the correct values are sent to the wrong place nothing will happen. These two concepts can be called generation and exploration. A good automated security testing tool tries to explore the target as much as possible whilst generating data which will trigger vulnerabilities.

MyFuzzer tries to solve both problems. Generation is done through the use of templates, which give structure to the data, and random mutations. The exploration part is solved by using a genetic algorithm aimed for increasing runtime code coverage. Note that the genetic algorithm is blind. It does not know the inner workings of the target, but using random mutation it will continue to find new areas of code. So both the exploration and the generation problems are taken care of in MyFuzzer.

1.4 Related Work

The idea of guiding a fuzzer using other techniques is not new. In [5] a concept for a fuzzer using a genetic algorithm is presented. Several other sources indicate the possible benefits of extending fuzzing with genetic algorithms [17, 21, 18]. Below are summaries of a selection of articles which tried to improve fuzzing one way or another.

In [8] it is proposed to extend fuzzing with dynamic taint analysis. The main idea behind their solution is to perform taint analysis on network packets and observe which sensitive functions the packets influence in the SUT. The main disadvantage of this method is that the fuzzer has to be told which sensitive functions to look for.

In [7] a solution which combines fuzzing, symbolic execution and taint analysis is proposed. Symbolic execution is used to find paths and valid variable assignments in the target. Taint analysis is executed on these paths to find taint information. The fuzzer uses the path information and path related taint information to generate test-cases. This solution is very complete but has some practical problems. One of the problems is how to implement a test-case generator which preserves path constraints.

In [5] another fuzzing solution is presented which uses taint analysis, con-colic execution, genetic algorithms and code coverage. The main steps of their method are:

1. Finding vulnerability patterns in the binary.

2. Perform taint analysis on the binary taking the found vulnerability patterns into account.

3. Test-case generation using concolic execution and search algorithms.

4. Use coverage information and genetic algorithms to generate next generation.

There are also some post-fuzzing steps but they are not relevant for this project. The disadvantage of this solution is that it is still very abstract. It raises questions about how the genetic algorithm is implemented and how code coverage is obtained.

In [18] static analysis and genetic algorithms are used to create a fuzzer called GAFuzzing. The static analysis is done using IDA pro. The output of the static analysis step is a set of vulnerability points or predicates, the control flow graph and an instrumented target. After the static analysis comes dynamic analysis. In the dynamic analysis step test-cases are generated and scored. The score of a test-case depends on the number of predicates it reaches. In [18] a tool called PIN is used. This tool is similar to the DynamoRIO tool used in this project. The disadvantage of this approach is that it still requires a static analysis step. MyFuzzer does not require this. On the other side the static analysis step might lead to better results. Another disadvantage is that this approach is dependent on the x86 processor architecture.

Some existing fuzzers need to be mentioned. First of all Sulley [27] is an open-source network fuzzer which uses attack libraries and a target specification to generate test-cases. Sulley evaluates the fuzzer results by using monitors on the target system. AFL [16] is an efficient and practical file fuzzer. It has found a lot of vulnerabilities in well known software. AFL uses a modified genetic algorithm to find good test-cases. The main difference between AFL and the fuzzer presented in this report is that AFL works on files whilst the fuzzer for this project works using network packets.

The main advantages of the solution presented in this report are that it requires minimal setup and that it does not require static analysis or pre-execution steps.

1.5 Report organization

The remainder of this report is organized as follows. In chapter 2 background information for this project is presented. The main topics are fuzzing, genetic algorithms and code coverage. In chapter 3 the analysis, design and implementation of MyFuzzer is given. Chapter 4 explains how the experiments were conducted. In the experiments chapter the experimental results are also displayed and analyzed. In the discussion chapter 5 the results are compared with other sources. Chapter 6 contains some pointers for future research and improvements of MyFuzzer. This report is concluded in 7 which will answer the research questions.

Chapter 2

Background

Some essential background needs to be provided before going further. The topics in this chapter are software security, software testing, coverage criteria, genetic algorithms and fuzzing.

Software security is included because the project is about making software more secure by finding vulnerabilities. The section about software security will explain why software security is generally so poor and what can be done about it. It also describes some common vulnerabilities. Knowledge about these vulnerabilities helps with developing a tool which can find them.

The section about software testing discusses two main topics about software testing: software security testing and test automation. Both topics are important because the focus of this project is on finding security defects automatically.

The topics of coverage criteria, genetic algorithms and fuzzing are included because MyFuzzer uses all three extensively. The last section contains some information on how these topics are combined in MyFuzzer.

2.1 Software Security

One of the most important components of cyber security and computer security is software security [20, chapter, p. 215]. This is because software is ubiquitous on computer systems and is used in every aspect of society. Software security is part of software engineering and has to do with developing secure software [20, chapter 1, p. 27]. These defects are important because they might be exploitable by attackers [20, chapter 1, p. 28]. Research has shown that the level of security in software has not increased over the years and there are reasons to believe that this will not happen in the near future. This has to do with three trends in software development. These trends are explained in more detail in section 2.1.2.

Software security aims to solve these problems by integrating security into the software development cycle. The definition for software security given in [20] can be seen in definition 1. Software security is not to be confused with application security or security software because those terms usually refer to tools which are applied after development.

Definition 1 *Software security is about understanding software-induced security risks and how to manage them. Good software security practice leverages good software engineering practice and involves thinking about security early in the software life-cycle, knowing and understanding common problems (including language-based flaws and pitfalls), designing for security, and subjecting all software artifacts to thorough objective risk analyses and testing. As you can imagine, software security is a knowledge-intensive field. [20]*

For this project software security is important. First of all fuzzing can be seen as a part of software security since it is used to identify software defects. Secondly software security is a knowledge intensive field. Some of this knowledge, especially the common vulnerabilities, are useful for this project since it helps to identify what we are looking for. Thirdly this section gives some basic insights about the importance of software security and why it is not getting better. The last part is not directly useful for this project but it does indicate that it is important to develop new techniques for software security.

2.1.1 Vulnerabilities

In [20] they talk about two types of vulnerabilities, implementation and design vulnerabilities. A definition of the term vulnerability as defined by [20] can be seen in definition 2, this definition matches with the definition given by the national institute of standards and technology [15]. The difference between the two types of vulnerabilities is that implementation vulnerabilities are simple mistakes like forgetting to check buffer sizes whilst design vulnerabilities are more fundamental mistakes. Vulnerabilities are important for this project since it tells us what to look for. Lots of known vulnerabilities are described in the literature. These common vulnerabilities are responsible for most software breaches [32, 24].

Definition 2 ***Vulnerability:*** *A defect or weakness in system security procedures, design, implementation, or internal controls that can be exploited and result in a security breach or a violation of security policy. A vulnerability may exist in one or more of the components making up a system.*

2.1.1.1 Defects

In [20, chapter 1, p. 37] definitions for the terms defect, flaw and bug are given. These can be seen in definitions 3, 4 and 5 respectively. The main difference between the terms are that a defect refers both to implementation and design vulnerabilities whilst a bug refers only to implementation vulnerabilities and a flaw only refers to design vulnerabilities.

Definition 3 ***Defect:*** *Both implementation vulnerabilities and design vulnerabilities are defects. A defect is a problem that may lie dormant in software for years only to surface in a fielded system with major consequence.*

Definition 4 ***Flaw:*** *A flaw is a problem at a deeper level. Flaws are often much more subtle than simply an off- by-one error in an array reference or use of an incorrect system call. A flaw is certainly instantiated in software code, but it is also present (or absent!) at the design level. For example, a number of classic flaws exist in error-handling and recovery systems that fail in an insecure or inefficient fashion.*

Definition 5 ***Bug:*** *A bug is an implementation-level software problem. Bugs may exist in code but never be executed. Though the term bug is applied quite generally by many software practitioners, I reserve use of the term to encompass fairly simple implementation errors. Bugs are implementation-level problems that can be easily discovered and remedied.*

Throughout this report the terms defect and vulnerability are used interchangeably. The defects MyFuzzer finds are most probably bugs since flaws are more difficult to find automatically.

2.1.2 Trends against Software Security

There are some trends which make software more vulnerable against misuse [20, chapter 1, p. 29]. These trends are connectivity, extensibility and complexity. These trends show why it is unlikely that software defect will be eradicated from software. In general the increase of vulnerabilities in software has to do with the fact that software becomes larger, more complex and more connected.

2.1.2.1 Connectivity

On first sight connectivity is a logical step in the evolution of software. Connectivity makes life easier (cloud storage, remote access, etc.) , however the downside is that the attacker has more possible entry points. An attacker needs only one weakness in one of the entry points to gain access. Another downside of connectivity is that an attacker can execute an attack from any location without getting caught.

2.1.2.2 Extensibility

With extensibility is meant that after the development of software new features can be added without rewriting the code. Examples of this are mods for games, extensions for the chrome browser and plugins for photoshop. Extensibility is useful because not all functionality needs to be available at release. A new feature can be introduced and installed with ease after a period of time. However one malicious extension might be enough to infect the whole system.

2.1.2.3 Complexity

The complexity of software is increasing with time. Software becomes larger and is dependent on the operating system, libraries and other dependencies. It has been stated by Bruce Schneier that complexity is the enemy of security [29]. There are some publications indicating the correlation between complexity and the number of software flaws. For example [12] published an article clearly indicating that an increase of lines of code leads to more security flaws.

2.1.3 Common Vulnerabilities

There are a lot of software vulnerabilities. However a few of them occur very often. Five of the most common vulnerabilities [32, 24] will be explained here.

Cross Site Scripting Cross site scripting, sometimes called CSS or XSS, is a vulnerability where input is interpreted as code by an application. In normal cases a user sends normal data to the application and it will work fine. However an attacker could send code to the application and the application would execute it. This vulnerability is very common in web-applications.

Buffer Overflow A buffer overflow happens when an attacker puts m bytes into a buffer which can only allocate n bytes, $m > n$. Usually these buffers are allocated on the stack. If there is a buffer overflow vulnerability then other parts of the stack can be overwritten. Usually this will result in a crash because the program will become corrupted. However carefully created input can make the program do what attacker wants to. A basic setup is this.

- Attacker finds a buffer overflow.
- The attacker determines the location of the return address and the buffer.
- The attacker puts malicious code in the buffer and overwrites the return address to point to the buffer.
- The malicious code will be executed.

This is a simplified example. There are techniques to prevent this like address randomization, non-executable stack and stack-protection, but there are also more complex attacks which take care of these protections.

SQL injection Like cross site scripting an SQL injection happens when data is executed as code. An SQL can occur when user input is used for creating an SQL query. If an input field contains this vulnerability a hacker can insert a malicious piece of SQL code which for example removes the underlying database.

Insecure Remote File Include Remote file include is an attack where the attacker supplies the server with a malicious formed file. For example if the server asks the client to choose a string 'a' or 'b' and consequently displays 'a.txt' or 'b.txt' then an attacker can try to send the string 'http://malicious.website/virus' as an option. If the remote file include is insecure than it will show 'virus.txt', given that the file exists on the web-server. More dangerous would be that the attacker sends the string '/etc/passwd' so that the server shows the users of the server and the password hashes.

Directory Traversal Directory traversal is best explained with an example. If a webpage can show the contents of files then it might be susceptible to directory traversal. If the URL for retrieving file content is “example.com/file=test.txt”, then you can try to access important files stored on the server by doing something like this: “example.com/file=../../../../../../../../../../../../etc/passwd”. If the server is susceptible it will show the contents of the password file.

Eliminating these common vulnerabilities from all software would increase software security enormously. An automated security testing tool should definitely try to find common vulnerabilities.

2.2 Software Testing

A common definition of software testing is: ”software testing is the process of checking if the implementation meets the requirement” [25]. Software testing is a component of the quality process [19]. The quality process spans through the whole development cycle of a product and has to make sure that the product meets the desired quality level [19]. The quality process consists of all activities which focus on checking the quality of the product in contrast to construction activities which have to do with building the product [19]. In context of the quality process, testing can be seen as the process for finding violations of the specifications in the product. The testing process spans multiple phases like planning the tests, generating test-cases, executing test-cases and evaluating test results. Documentation and system requirements are used to verify the test results. Lack of documentation has a negative influence on software testing because without documentation it is hard to verify that the software meets the requirements of the client [19].

Testing has two main aspects, validation and verification [19, 25, 26]. Verification has to do with making sure that the implementation matches with the requirements or specification [19, 26]. On the other hand, validation of a software system is making sure that the user expectation matches with the implementation [19, 26]. The difficulty of software testing is that the requirements are not always fixed or open to multiple interpretations. For example the requirement ”the software should be fast” is not very specific and therefore difficult to test. The requirement should be made more clear by quantifying what fast is. Some common terminology in software testing [26]:

Test Case A test case consists of test case values, which are the values needed to form an execution in the system under testing. In addition to the test case values a test case consist information about expected results.

Test Suite Also called a test set. Is a set of test cases. It is a good habit to keep test suites structured.

Testing can be done on several levels, from system wide to individual code units. Each level has its own testing techniques and needs a different approach. In [26] five levels are presented, from low to high:

Unit Testing Unit testing tests individual elements of the system. It has to do with verifying if the implementation matches the expected functionality. A unit is a coherent set of code, like a function or class.

Module Testing Modules can be seen as a bundle of units. Module testing takes a module and checks if the implementation matches its requirements. The only difference with unit testing is the scope.

Integration Testing Integration testing works on the subsystem level. It checks if the modules of a subsystem work together in a consistent way. In integration testing it is assumed that the modules are correct.

System Testing In system testing it is verified that the system meets its requirements. It assumes the lower level components to work, which can be verified with the previous mentioned testing levels. System testing does not test the implementation, it tests the analysis and design of the system.

Acceptance Testing Acceptance testing is validating with the client that the system does what the client wants.

Software does rarely stay the same. New updates, patches and configuration changes might break some parts of the system. Regression testing aims to solve this by retesting the system after each update to make sure it still functions properly. If the system is large it can be more efficient to perform regression testing on a subset of the system instead of retesting the whole system. Note that these testing techniques are usually executed at different times and test different aspects of the system. Module and unit testing test the implementation whilst system testing tests the analysis and design of the system [26]. Testing can also focus on different aspects of the system like usability, robustness, security and functionality [25].

2.2.1 Limitations

Software testing has some limitations. The most important limitations are:

Undecidable In general it is undecidable if all failures of the SUT have been found [26]. Usually this limitation is accepted and the tester tries to find test-cases which fail. There are also methods which uses over- and under-estimation in order to give guarantees about the vulnerabilities in the SUT. For example a sound static analysis tool gives the guarantee that the set of all failures in the SUT is a subset of the vulnerabilities it found, although there might be false positives.

Complexity Software becomes increasingly more complex and thus it becomes harder to find software defects using testing methods [25].

Because of these limitations some alternatives to testing have been proposed like code reviews and formal methods. These methods have limitations as well.

2.2.2 Software Security Testing

Security testing limits the scope of testing to the security properties of a program. There are two important types of security testing. The first one is to test explicit security requirements which is called security functional testing. Security function testing tests the functionality of programs which have to do with security like authentication, integrity and confidentiality. So in this case only parts are covered which have to do with security implementation. The second type is vulnerability testing which tries to discover vulnerabilities in a program. A vulnerability is defined as a part of a system which potentially can be used in other ways than it was intended [22]. There consist several methods for doing security vulnerability testing like formal security testing, model based security testing, fault injection based security testing, fuzz testing, vulnerability scanning testing, risk based security testing and white box security testing [22]. MyFuzzer is an instance of security vulnerability testing.

2.2.3 Test Automation

A lot of research is done on test automation [26, 28]. The benefits of test automation is that parts of the testing process are automated and thus take less time. Some other benefits of automated testing are that it reduces human errors in the testing process and regression testing becomes easier.

A specific part of test automation is test-case generation. Test-case generation is the process of generating a set of test-cases given a set of test requirements [26]. Writing test-cases manually is already difficult for a tester because he has to write meaningful test-cases. A tester writes test-cases based on the source-code, if it is available, past knowledge and test requirements. In short the tester uses his intellect to construct test-cases. If the test-cases are generated the algorithm needs to come up with meaningful test-cases as well. This is difficult but there are solutions to this. In [28] five techniques are explained:

- Symbolic Execution.
- Model-based Test-case Generation
- Combinatorial Testing
- Adaptive Random Testing
- Search-based Testing

Fuzzing is another technique for test-case generation. Fuzzing is explained in 2.5. As stated in [26] it is important to note that these techniques can be combined to form different automated testing solutions.

2.3 Coverage Criteria

The set of possible inputs I for a given SUT S is usually very large. For example take a program which takes as input a string s and an integer i, where $\#s < 20$, then there are already $2^{32} * 256^{20} = 2^{32} * 2^{160} = 2^{192}$ inputs. It is already unfeasible to test every input for SUT S, and a program has usually way more than two inputs. Coverage criteria are used to solve this problem by deciding which inputs to use. A coverage criterion is defined in terms of test requirements. A definition of a requirement is given in 6.

Definition 6 *A test requirements is a specific element of a software artifact that a test-case must satisfy or cover. [26]*

Test requirements can be given on different levels. For example a requirement might be ”every line of code needs to be tested at least two times”, whilst another requirement might be ”every node in the class diagram must have at most three connections to other nodes”. The first is on the implementation level whilst the other is on a design level. A coverage criterion is a recipe for generating test-cases [26]. So from a coverage criterion C a set of test requirements TR can be generated. A test-set T is measured in terms of coverage. Coverage is achieved if T satisfies every test requirement for a coverage criterion C. Usually testers make use of a coverage level. Coverage level is defined as the ration between the test-requirements satisfied by T and the size of the set of test requirements. So if there are five test requirements for coverage criterion C and a test-set T only satisfies three test requirements the coverage level is $3/5$. Usually the term coverage level is omitted and it is simply stated as a coverage of $3/5$.

The main difficulty with coverage is to construct a complete and relevant set of test requirements.

On implementation level there is code coverage, which counts as a subset of the general test coverage [26]. Code coverage has to do with keeping track of which parts of the code or system have been tested and which parts have not [19]. This is useful in locating regions which might need additional testing. A good testing method should cover all the relevant parts it wants to test, no relevant region should be left untested in the optimal case. Code coverage is said to be white-box in that it takes the program code into account [9]. Black-box or gray-box code coverage is however also possible by doing code coverage on assembly code. To do this some reverse engineering techniques, runtime executable manipulation software and memory scanners [2, 13] might be useful. Code coverage has multiple metrics but the general idea is to indicate how much of the source code has been tested. In [9] they talk about code coverage analysis, this is a testing method which uses code coverage as instrument. There are several code coverage metrics, defined on multiple levels [9]:

Statement Coverage Statement coverage counts every executable statement which was executed. This excludes control statements. The advantage of this metric that it is easy to calculate. On the negative side, it does not take into account control flows and logical statements.

Decision Coverage Decision coverage checks if every Boolean expression in a control statement has been evaluated to both true and false. The rationale behind this is that with decision coverage you can check if every distinct path has been tested.

Condition Coverage Condition coverage is like decision coverage the only difference is that it is verified if every condition is evaluated to both true and false. In decision coverage only the outcome of the whole expression is evaluated. Full condition coverage does not mean full decision coverage, for example the statement “if a and not a: statement” cannot have full decision coverage but can have full condition coverage. Control coverage has better sensitivity to the control flow than decision coverage.

Multiple Condition Coverage Multiple condition coverage is an extension of condition coverage. Where condition coverage only looks at atomic Boolean statements individually, multiple condition coverage checks if for each expression all possibilities have been evaluated. So if you have the expression $a \wedge b \wedge (c \vee d)$ you have full multiple condition coverage if all possibilities of a, b, c, d have been evaluated. Note that if the programming language uses short circuit logical operators that the number of possibilities can be decreased. In that case the possibilities would be $a \mapsto F, a \mapsto T \wedge b \mapsto F, a \mapsto T \wedge b \mapsto T \wedge c \mapsto F \wedge d \mapsto F, a \mapsto T \wedge b \mapsto T \wedge c \mapsto T \wedge d \mapsto F, a \mapsto T \wedge b \mapsto T \wedge c \mapsto F \wedge d \mapsto T, a \mapsto T \wedge b \mapsto T \wedge c \mapsto T \wedge d \mapsto T$. Multiple condition coverage can be very thorough.

Path Coverage Path coverage considers if every path through the code has been executed. With a path is an unique sequence of commands from start to end. Path coverage has some difficulty with loops because they increase the number of paths which can even lead to an infinite number of paths. This can be solved by using a boundary-interior path coverage which reduces loops to binary variables, zero runs or more then zero runs. Some other disadvantages are that path coverage increases exponentially with the number of decisions or branches, also some paths are impossible due to data restrictions.

This list is not conclusive, [26, 9] identify a lot more code coverage metrics. More about coverage criteria can be found in [19, 26, 9].

2.3.1 Importance to Software Testing

Code coverage is important for software testing because code coverage gives measurements about how much of the SUT has been tested. These measurement can be used to say something about the quality of the tests. For example if there are two test-sets for the same SUT called test-set a and test-set b, where test-set a has a statement coverage of 0.6 and test-set b has a statement coverage of 0.9, then you can conclude that test-set b is better. This conclusion is not necessary correct but the statement coverage metric is an indication. If several code coverage metrics are used in combination with other test requirements like each branch, line of code or function needs to be tested at least n times then we can argue about the quality of the test-set. Therefore code coverage is important to software testing.

2.4 Genetic Algorithms

A genetic algorithm is a searching method which can be used to find near-optimal solutions in a large solution-space [31]. It is a method based on natural selection where selection, mutation and crossover are important concepts [4]. Genetic algorithms work by creating a random population consisting of several individuals which is called generation 0. The individuals in the population receive a score based on a fitness function. Based on the fitness scores a subset of the population is selected for the next generation. The higher the individual's fitness the higher the probability it gets selected [4]. The selected individuals get recombined and mutated to form the next generation [4]. This process continues until the stop criterion is met. Some common stop criteria are [4]:

1. A maximum number generations.
2. A certain amount of time.
3. The solution has been found

More formally a genetic algorithm starts with a randomly selected population $P_0 \subset X$, where X is the set of all possible individuals. Each $p \in X$ consists of n genes $p_0, p_1, ..., p_n$. A gene is a small part of information which encodes a property of the individual. Each $p \in P_0$ is evaluated according to a fitness function $f(p) = s$, where the higher the score the higher the fitness. If f is applied to all $p \in p_0$ then we have a set $S_0 = \{s_0, s_1, ..., s_{\#P0}\}$. Based on the scores a selection $C \subset P_0$ is taken where the highest scores have the highest chance of being selected and the lowest scoring individuals have almost no chance being selected. Scoring can be time consuming, to take care of this sometimes only a random sample is scored [33]. The next step is apply genetic operators on the selection C. Usually the set op genetic operators consists of a mutation operator M which transforms a individual i to i' and a cross-over operator C_O which given two(or more) individuals $i_1, i_2, i_1 \in C, i_2 \in C$ generates a new individual i_c. Other operators are possible but these are the most common [33]. The rates at which these operators are applied are important. Too few mutations may lead to genetic drift, too much mutations can lead to solutions being thrown away. Too much re-combination may lead to a too early convergence [33].

For this assignment the individuals of the population are test-cases each test-case consists of a set of genes where a gene can be identified as a piece of data like a string, delimiter or integer

2.4.1 Operators

Like explained earlier, genetic algorithms consist of three main operators: selection, cross-over and mutation. It is important to explain every operator in more detail because they are used for MyFuzzer quite extensively.

2.4.1.1 Selection

Selection is the process of selecting a sample from a population. There are several selection mechanisms. Each mechanism works differently, but usually the individuals with the highest fitness score have the highest chance of being selected. Selection needs to be tuned together with crossover and mutation in order to get a well functioning GA [23]. Too much selection leads to early convergence(no more variance) and to slow selection leads to a slow evolutionary process [33, 23].

2.4.1.2 Crossover

Crossover is the process where n or more individuals are used to create a new individual i. The simplest form of cross-over is single-point cross-over where, given two parents i_1 and i_2, a point $p, 0 \leq p \leq l$ is randomly selected. Let $i(k), 0 \leq k < l$ be the $k'th$ bit of i, then the new individual i is constructed as follows $i(k) = i_1(k),\ if\ k \leq p$ and otherwise $i(k) = i_2(k)$. So all bits below p are selected from i_1 and the rest is selected from i_2. Single-point crossover has several shortcomings: it cannot combine all individuals and suffers from positional bias [23]. There is also double-points cross-over which extends single-point cross-over by selecting two points p_1 and p_2. An individual is than constructed as follows $i(k) = i_1(k),\ if\ k \leq p_1$, $i(k) = i_2(k),\ if\ p_1 < k \leq p_2$ and $i(k) = i_1(k)$. Another crossover scheme is 50-50 crossover where each bit has 50 percent change of being selected from i_1 and 50 percent change of being selected from i_2. It is noted in [23] that it is unclear what cross-over method is the best.

2.4.1.3 Mutation

Mutation is the third main part of a genetic algorithm. Mutation takes an individual i and flips bits at random locations. Mutations can be more complex than bit-flips. The general idea is that a mutator takes an individual i and creates a mutation i_m

which is slightly different from i. There is some debate about whether cross-over and mutation are both necessary since they seem to do the same thing, which is adding variance within the population. Research shows that cross-over is more robust than mutation but also that mutation helps in the process [23]. The key is to find balance between mutation and crossover.

2.4.2 Test-case generation

The relevancy of genetic algorithms to automated test-case generation is that a genetic algorithm could be useful in searching the input space. If a proper fitness function can be defined the GA can select the best test-cases and throw away the others. Using mutations and cross-over the search space could be explored in a structured way.

In [1] it is indicated that fuzzing could be improved by using genetic algorithms. A genetic algorithm has also been used in the fuzzer AFL [16].

Genetic algorithms are relevant to the subject because they help with searching. As described before, there are almost infinite possibilities for test-cases and not all can be evaluated. A genetic algorithm would help finding good test-cases while neglecting the bad test-cases. The basic concept of combining test-case generation with a genetic algorithm would look like this:

1. Generate a first generation of test-cases.
2. Send test-cases to the SUT and measure the results.
3. Based on the results, calculate the fitness per test-case.
4. Use the fitness scores to make a selection.
5. Perform crossover and mutation in order to create the next generation.
6. Repeat.

A difficulty is how to define the fitness of a test-case. In AFL a test-case gets a higher fitness score if it tests new parts of the SUT. An other idea is to make the fitness dependent on the number of variables changed or accessed.

2.5 Fuzzing

Fuzzing is defined as follows: ”Fuzzing is an interface testing method aiming to detect vulnerabilities in software without access to application source code.“ [5]. Another common definition for fuzzing is: ”Fuzzing is a security testing approach based on injecting invalid or random inputs into a program in order to obtain an unexpected behavior and identify errors and potential vulnerabilities“ [5]. What can be learned from these definitions is that fuzzing aims to test a system by sending lots of (semi)random inputs [31]. The advantage of fuzzing is the ability to generate lots of test-cases in a short time. On the other hand fuzzing does not guarantee that the generated test-cases will find all vulnerabilities. It also does not give any guarantees about the quality of the generated test-set. Fuzzing has been used to find a lot of security flaws in software and network protocols [31, 16].

2.5.1 Generation and Mutation Fuzzing

Fuzzers can be distinguished into two types, mutation and generation fuzzers [11]. Mutation fuzzers take existing input for a system and randomly mutate parts of the data each cycle. If X is the set of possible inputs for the SUT, then a random fuzzer will generate an input $x, x \in X$ and send it to the SUT. The generation process starts with given inputs I, where each input $i, i \in I$ can either be valid or invalid input. Let $T(n)$ be a function which returns the n'th test-case then $T(0) = i, i \in I$ and $T(n) = M(T(n-1)), M \in MU$, where MU is a set of transformation functions which perform a transformation on test-case $n-1$. M can be multiple functions. This processes is repeated until a stop criteria $c \in C$ is met, where C is the set of stop criteria. Stop criteria could be a maximum number of test-cases m, a time-span t or if the SUT crashes [31]. Some tools which make use of mutation fuzzing are zzuf and AFL[6, 16]. Generation fuzzers on the other hand have knowledge about the protocol or file structure and use this to generate (semi)valid messages. A generation fuzzer does the same as a mutation fuzzer with the constraint that the set of all inputs I match with the given structure. An example of a generation fuzzer is peach [3], although it also has mutation capabilities. In real scenarios generation fuzzers are to be preferred because they generate valid inputs, whereas a mutation fuzzer might not always generate valid input [11]. Another example of a generation fuzzer is Sulley [27]. Note that the two types are not mutually exclusive, but that a generation fuzzer can use mutation fuzzing. Also note that not all fuzzers are the same. Some fuzzers use non-random methods for fuzzing as well.

Both types of fuzzers still suffer from the same problem that it is not guaranteed that it finds all paths through the target.

As can be seen in the solution chapter 3, MyFuzzer is mainly a generation fuzzer.

2.5.2 White-box versus Black-box Fuzzing

Like normal testing, there is black, white and gray box fuzzing [5]. Black box fuzzing does not have any knowledge about the system. White box fuzzing does have a complete knowledge about the system and gray box fuzzing has limited knowledge about the system.

2.5.3 Fuzz Vectors

Fuzz vectors are used to make fuzzing more directed and less random. A fuzz vector is a string which contains a common dangerous value, for example a cross-site scripting value. These fuzz vectors are then mutated and inserted into the system. Fuzz vectors can help with finding vulnerabilities a lot if the vectors are representative of common vulnerabilities. Fuzz vectors are used in the solution presented in this report.

2.5.4 Guided Fuzzing

The random element of fuzzing is not always an advantage. That is why it is a good idea to guide the fuzzing process into a certain direction. By guiding the fuzzing process the tester can increase the quality of the tests. Data is collected for every test-case sent from the fuzzer to the SUT and based on the data the test-case is assigned a fitness score. In [16] the number of new transitions between basic blocks are used in order to score a test-case. Code coverage in general seems a good metric for guided fuzzing. However other metrics, like the number of changed variables and memory scanning might also be possible. One difficulty is that in order to collect these metrics usually re-compilation of the SUT is required. There are methods for which this is not necessary however these tools decrease the performance. On the upside these tools do not need source-code so it can be applied for all binaries. For re-compilation there are tools like gcov to collect coverage data. Without re-compilation there are several run time code manipulation systems like QEMU, DynamoRIO and PIN which can be used for data extraction.

2.6 Putting Things Together

The topics described earlier in this chapter can be used to construct a smart fuzzer. A smart fuzzer is a fuzzer which uses information to generate better test-cases. The basic idea can be described as follows:

Fuzzing is undirected, there are no quality guarantees about code coverage and finding vulnerabilities. A genetic algorithm can be used to guide the fuzzing process to a goal. A reasonable goal is to have a high code coverage. Although good code coverage does not necessarily mean that the test quality is high, it is used as test quality metric in the testing community [26]. Code coverage can be maximized to give test-cases which find new code a higher score thus giving it more chance to stay in the population. So the basic idea is to generate random test-cases in the beginning, evaluate the impact of the test-cases on the target in terms of code coverage and then use a genetic algorithm to construct a new set of test-cases which consists of mutated versions of the highest scoring test-cases.

More formally it can be written like this. Given a target T there is a set I which denotes all possible inputs or test-cases for T. Consider the function $V(i) = x, x \in \{0, 1\}$ to be a boolean function which indicates if test-case i triggers a vulnerability or not. The main goal of the fuzzer is to find inputs $i, i \in I$ where $V(i) = 1$. It is assumed that code which has not been tested has more probability of containing a vulnerability, so the genetic algorithm tries to guide the fuzzer towards untested pieces of code. For the first set of test-cases a random sample S of I is taken. Each $s, s \in S$ gets a score after being sent to the target T. The score is based on code coverage metrics, which are based on how much new code a test-case has found. The genetic algorithm selects a set of test-cases S' from S based on the scores. Then the genetic algorithm uses cross-over to fill S' to its original size. After the cross-over step S' is mutated and the process is repeated. This keeps on repeating until an appropriate stop criterion is met.

2.6.1 Example

A demonstration of how this approach works can be seen in the following example. Consider the following piece of pseudo-code.

```
run(char *data, int seed)
{
        char *buffer = malloc(sizeof(char) * 20);
        Random rand(seed);
        int x = rand.random(6);
        int y = rand.random(6);
        int z = rand.random(6);
        int sum = x + y + z;
        if(sum == 18)
        {
                strcopy(data, buffer);
```

```
          }
          else
          {
                    // nothing happens
          }
}
```

The if statement is expected to be executed once every $6^3 = 216$ times. In normal fuzzing a seed which triggers the if statement might be ignored in the next round. A fuzzer which uses a genetic algorithm to optimize code coverage will give test-cases which trigger the if statement preference over other test-cases and so it is more likely that it will find the buffer overflow in the strcopy function.

Chapter 3

Solution

This chapter describes the design and implementation of MyFuzzer. It consists of several sections. The first section gives the requirements for MyFuzzer. The second section provides an overview of the solution and gives some reasons why this solution is an improvement over other solutions. The design section explains the static and dynamic design of MyFuzzer. The static design explains the architecture of MyFuzzer whilst the dynamic design shows the interactions between components. The implementation section contains some low-level information about how MyFuzzer was implemented.

The solution presented in this report is a proof of concept. Before using it in practice it should receive some revisions. These revisions are needed to make the tool more usable in practice are described in the discussion and future work chapters 5 and 6.

3.1 Requirements

The assignment was flexible, so at the start of this project there where not a lot of hard requirements. The initial assignment was the development of an automated security testing framework which could be configured and extended with new modules later on. Eventually it was decided to use fuzzing as a solution. Through interviews and meetings with employees from Thales and the University of Twente additional requirements were obtained. The requirements are split up into functional and non-functional requirements. The requirements should give an overview of what MyFuzzer is supposed to do.

3.1.1 Functional Requirements

MyFuzzer should at least contain the following functionality:

1. MyFuzzer has to be able to understand how to communicate with the target. This can be split up into two sub-requirements.

   (a) MyFuzzer should have the means to communicate with the target over IP using UDP and/or TCP.

   (b) Given the target communication specification, which is a complete description of how the fuzzer can communicate with the target, MyFuzzer should be able to construct valid messages.

2. MyFuzzer needs functionality for genetic algorithms. This requirement can be split up in several sub-requirements.

   (a) MyFuzzer should be able to generate a set of inputs according to an input specification. The set of inputs should match the input specification but also be sufficiently different from each other.

   (b) MyFuzzer should have capabilities for mutating data

   (c) MyFuzzer should be able to recombine two or more inputs into one new input (cross-over).

   (d) MyFuzzer should have a fitness function which can score a piece of data.

3. MyFuzzer should be able to monitor targets to detect crashes and obtain debug information.

3.1.2 Non-functional Requirements

The following non-functional requirements where identified. Stability/robustness is by far the most important requirement. A fuzzer can trigger strange behavior in the target, target monitor and even the fuzzer itself. So MyFuzzer should be able to recover itself in those cases.

Stability MyFuzzer has to generate and execute a lot of test-cases. This has to be done automatically, so it is important that MyFuzzer keeps running. A fuzzer is a complex system so achieving stability might be a challenge. To make the system stable it should have proper exception handling and should contain handles for restarting parts of the system.

Efficiency It is important that MyFuzzer sends out test-cases frequently. The efficiency is platform dependent but in general MyFuzzer should be able to execute a test-case in less than a few seconds.

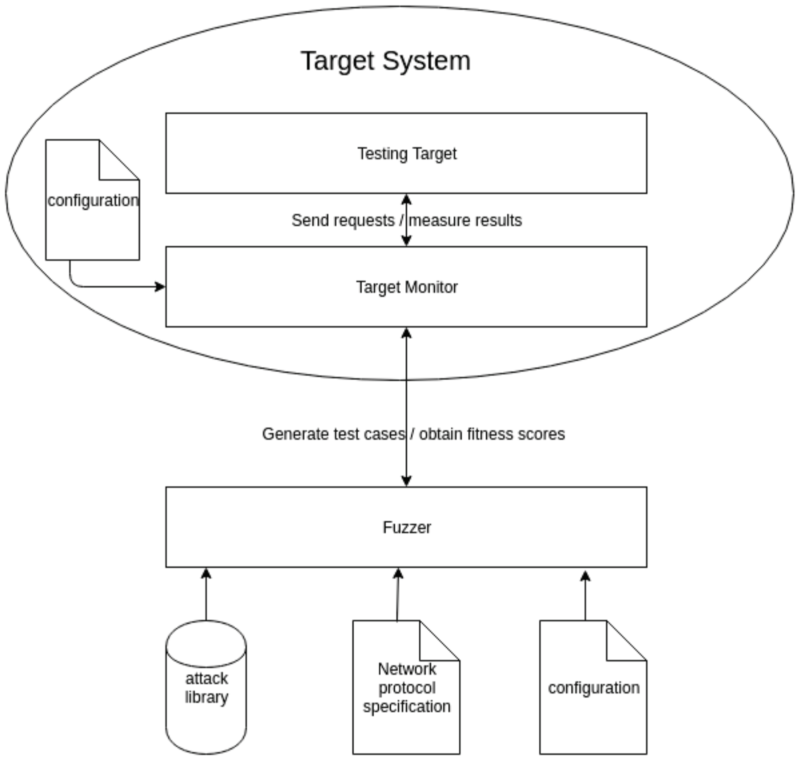

Target System
Testing Target
configuration
Send requests / measure results
Target Monitor
Generate test cases / obtain fitness scores
Fuzzer
attack library
Network protocol specification
configuration

Figure 3.1: An overview of MyFuzzer

Automation This requirement is trivial because the whole idea of a fuzzer is to automate testing. However it is good to state again that the solution presented should be able to do its work automatically for 99 percent of the time.

Practical MyFuzzer should work on most network applications without much setup or other difficult steps. It also should work without needing a lot of information like source-code and flow diagrams.

3.2 Overview

MyFuzzer is a gray-box network fuzzer which uses a genetic algorithm to search for good test-cases. This is different than more traditional random fuzzing, which sends random test-cases without evaluating the results, or deterministic fuzzing which generates test-cases by trying all possibilities.

MyFuzzer has several modules, which can be seen in figure 3.1. As can be seen MyFuzzer consists of three modules, the fuzzer, the target and the target monitor. Each module will be explained in more detail.

Target The target is also known as the system under testing, so the target is the application which is being tested. For this project it is assumed that the target can consists of multiple applications but they are all running on the same system. Adding distributed systems is however a simple extension so it could be added in future releases of MyFuzzer. The target is a network application, which has an ip address and port number. Clients can communicate with the target by sending requests to this ip address. The fuzzer is a client so it communicates with the target using ip. The target monitor runs on the same system as the target and identifies the target by its process id.

Target Monitor The target monitor monitors the target. It checks the target for crashes and malicious behavior like memory corruptions. The monitor also measures the impact of a test-case on the target. These measurements are sent to the fuzzer which in turn gives it to the genetic algorithm to score the test-case. The target monitor is tied to the fuzzer. Without the target monitor the fuzzer would not be able to detect crashes and evaluate the quality of test-cases.

Fuzzer The fuzzer generates the test-cases which are sent to the target. The initial set of test-cases is generated randomly by the fuzzer. After that the genetic algorithm is used to generate new sets of test-cases. This is done by selecting the best test-cases from the set based on code coverage metrics, and letting the best test-cases create offspring. The set is than mutated to create the new set of test-cases. The genetic algorithm is an important component of the fuzzer and will be explained in the next section 3.3. The fuzzer also communicates with the target monitor to obtain the measurements for the sent test-cases.

The target monitor and the fuzzer together constitute MyFuzzer. The interactions can be divided in to three main steps: initialization, execution and evaluation. The initialization step reads the configuration files and makes everything ready. The execution step starts with the generation of a set of test-cases according to the configuration and uses the genetic algorithm to generate new sets of test-cases. The genetic algorithm uses the results obtained from the target monitor. The evaluation step is about the evaluation of the results. This step should answer questions like:

- How many errors have been found.
- How long did it take.
- What is the code coverage.

In the next section the design of MyFuzzer will be explained in more detail.

3.3 Design

This section explains how MyFuzzer is designed. After reading the information in this section the following things should be clear:

- How the genetic algorithm is combined with the fuzzer.
- How the fitness of a test-case is calculated.
- Which code coverage metrics are collected.
- How test-cases are generated and mutated.
- How the target communication specification is given and used by the fuzzer.
- Why there are multiple fuzzing strategies and how they differ from each other.
- How and when components interact with each other.

It has to be said that the implementation does not always match the design. This is due to the fact that the implementation of MyFuzzer is based on Sulley. The consequence of this is that the implementation of Sulley does not always match the design presented in this section.

3.3.1 Architecture

The architecture of MyFuzzer can be seen in figure 3.2.

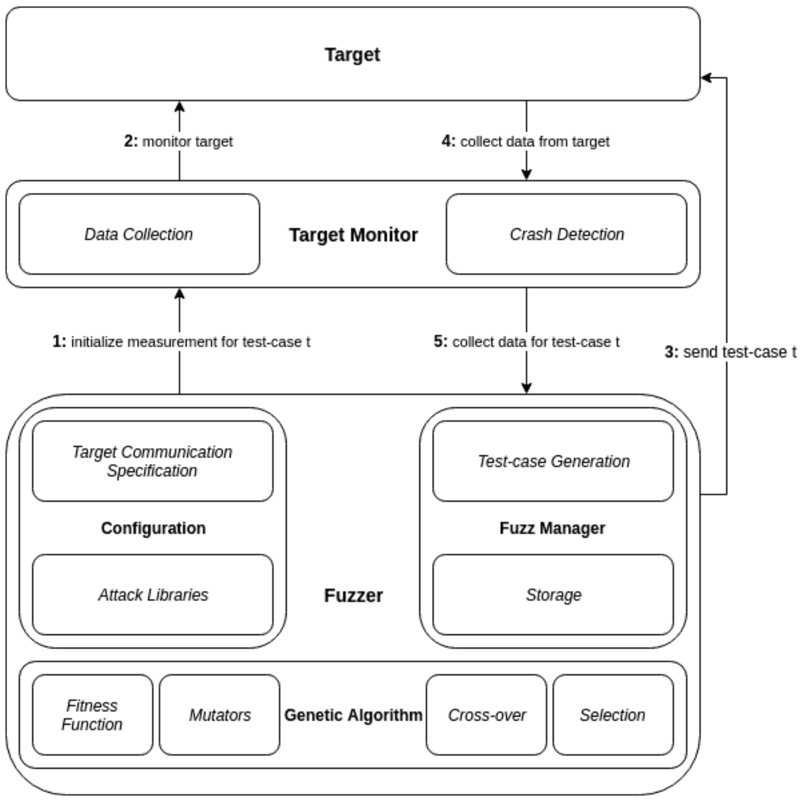


Figure 3.2: The architecture of MyFuzzer

As can be seen in figure 3.2 MyFuzzer consists of multiple components and sub-components. Every component will be explained in more detail in the following sections.

3.3.1.1 Fuzzer

3.3.1.1.1 Configuration Configuration is the label applied to all functionality which is used by the fuzzer to be able to start fuzzing. So the configuration component has the information to let the Fuzz Manager know where the target is located and how the requests are structured. It also tells the GA the importance of each coverage metric, which has influence on the fitness function, and which mutation rates the GA has to use. The concepts of fitness function, mutation rates and coverage metrics will be explained in sections 3.3.1.1.3 and 3.3.1.2. The configuration consists of several important sub-components.

3.3.1.1.1.1 Target Communication Specification MyFuzzer is a fuzzer for network applications, which means that the fuzzer communicates with the target through a network(ip and port number). Network applications can be seen as a state diagram with states and transitions. A transition can be seen as a command which brings the application from one state to another. Note that the state could be the same e.g. after doing command x the application went from state 1 to state 1. For example consider an imaginary authenticated FTP server application. In that case there are two states, *AUTH* and *PRE-AUTH*, where the latter is the default state. To use all FTP commands you have to be authenticated, so, assuming there is a transition between *AUTH* and *PRE-AUTH*, you would have to issue a command which goes from *PRE-AUTH* to *AUTH*. For our fuzzer the commands are issued by sending messages to the target over the network. For example the state transition mentioned earlier could be triggered by the *AUTHENTICATE* command, so the message sent to the target could be something like *AUTHENTICATE USERNAME PASSWORD*.

So the target communication specification has two components. First of all it contains a list of templates, one for each distinct message. These templates describe how a message has to be constructed. The second part of the target communication specification is to describe the states and transitions between states.

For MyFuzzer the list of all possible messages are given in a block structure. A message can consist of several blocks and each blocks in term can have several other blocks. Eventually a block is reduced to a primitive or a set of primitives. Primitives are simple pieces of data like strings, integers, set of commands, delimiters or random data. Using this block structure almost every possible message can be constructed. For example the message "AUTHENTICATE USERNAME PASSWORD" can be represented as the template *[COMMAND][DELIM][STRING][DELIM][STRING]*. In this template there is only one block and this block consists of five primitives.

3.3.1.1.1.2 Attack Libraries Attack libraries are collections of vulnerable inputs. Each primitive can have an individual attack library. For example the string attack library contains several potentially dangerous strings, like SQL injections and string format attacks. Attack libraries are used when an input is mutated, with a certain probability the current value is replaced by an input from the attack library.

3.3.1.1.1.3 Execution Parameters The execution parameters instruct the fuzzer and genetic algorithm how to do the fuzzing. There are several parameters which can be specified, the most important being:

Strategy Specifies the fuzzing strategy which MyFuzzer has use. There are four strategies which are explained in more detail in section 3.4.6.

Mutation Rates Mutation rates are tied to a piece of mutatable data. It specifies the likelihood of a mutation happening. Per mutator a mutation rate has to be specified. Mutators and mutation rates will be explained in more detail in section 3.4.

Coverage Metrics Weigths Coverage metrics weights, also called coverage weights, specify the importance of a coverage metric. A higher weight indicates that the metric is more important and thus it will have more impact on the fitness function.

3.3.1.1.2 Fuzz Manager The fuzz manager is responsible for storing and executing the test-cases. It also keeps track of all aspects of the fuzzing process and relies on the configuration and genetic algorithm components. The fuzz manager is also responsible for test-case generation.

3.3.1.1.2.1 Test-case generation Although generation has a lot to do with the genetic algorithm, test-case generation is seen as a sub-component of the fuzz manager. Test-case generation takes the Target Communication Specification as input and constructs a set of test-cases which match the specification whilst randomizing the data itself. So the output is a set of test-cases which have the same structure but contain different data.

3.3.1.1.3 Genetic Algorithm The genetic algorithm is the component which does contain the functionality for mutating data. It also contains the functionality for scoring test-cases, selecting test-cases based on score and creating new test-cases from a set of test-cases. Like explained in the literature section these are the basic functions of a genetic algorithm. In the case of fuzzing the individuals are test-cases. There are two types of genetic algorithm implemented in my fuzzers. The firsts GA implementation follows the literature strictly, so it generates a population of test-cases, scores all test-cases, takes a selection of test-cases and uses them to create new test-cases. After that the test-cases are mutated. This is different from the second implementation which relaxes the requirements from the literature a bit. In the second implementation a queue with test-cases is generated. After sending a test-case from the queue the results are compared to thresholds. If the scores are higher then the thresholds the test-case will be mutated and added to the queue again. At some times a cross-over step is done to create new test-cases. In MyFuzzer this is done when there have been n consequent test-cases which did not meet the thresholds. So in the second implementation there is no such thing as populations. It was implemented because some fuzzers use an implementation like this and had quite a lot of success with it [16]. Another reason is that it makes the process easier, since there is no need to evaluate groups of test-cases anymore.

The sub-components of genetic algorithm are explained in the next sections. Note that for the second implementation the selection component is not used. Mutation, cross-over and the fitness function are however used, although be it in different forms for both implementations.

3.3.1.1.3.1 Fitness Function The fitness function scores test-cases based on eleven metrics, which will be explained in a later section. The higher the score the more chance of being selected in the selection step. Each test-case gets a score for each of the eleven metrics after it has been executed. These scores are normalized with each-other to make each metric equally important. Each metric also has a weight associated with it. The metric weight determines the importance of the metric. For example a metric with metric weight 0.8 determines for 80 percent the resulting fitness score. The fitness function can be defined formally as follows : $f(x) = w_1 * m_1(x) + ... + w_{11} * m_{11}(x)$, where x is a test-case, w_i is the weight for metric i and $m_i(x)$ is the value for metric i for test-case x. Also note that $\sum_{i=1}^{11} w_i = 1$ and that $\forall_{w \in \{w_1,...,w_{11}\}} 0 \leq w \leq 1$.

3.3.1.1.3.2 Selection The selection step does the following. From a population of size n, where each test-case in the population has been scored, $n/2$ test-cases are selected. This is done using a random weighted select. What that means is that test-cases with higher weights(which are the scores they received from the fitness function) have more probability of being selected. The selection is important because it roots out bad test-cases and keeps the best ones.

3.3.1.1.3.3 Crossover Crossover is explained in the background chapter 2.4. Cross-over happens after the selection step. The selection step reduced the number of test-cases by a half. The goal of the crossover is to increase the number of test-cases to the original value. This is done by taking two random test-cases from the available test-cases and use them to create a new test-case. This happens until the number of test-cases is the same as before.

MyFuzzer uses simple 50-50 cross-over which means that a new test-case receives a piece of data with 50 percent from parent 1 or with 50 percent from parent 2. There are two things which make crossover a bit more complicated. First of all there are exceptions where parent 1 and parent 2 do not share the same template. This is solved by determining which pieces of data the parents do not have in common. The child inherits a piece of data d which is only available in one parent with 50 percent change, otherwise it is omitted. The second complication is how deep the crossover has to be done. In the section about the target communication section 3.3.1.1.1.1 it is explained that blocks can be nested. Currently MyFuzzer performs crossover only on the top level. So if the template for both parents is *[COMMAND][BLOCK][STRING]* then *BLOCK* is inherited entirely from parent 1 or parent 2. Alternatively crossover could be performed on *BLOCK* itself because it consists of primitives and other blocks as well.

In the implementation section 3.4 cross-over is explained in more detail.

3.3.1.1.3.4 Mutation Mutation happens after the crossover step. The goal is to create variance within the population. Without the mutation step the inputs would look too much like each other. The mutation step is executed on the highest level, usually a request, which propagates to lower levels and eventually the primitives. The primitives contain concrete implementations for mutation. The mutations per primitive are given in the implementation section 3.4.

3.3.1.2 Target Monitor

The target monitor has several important functions:

1. Monitor the impact of a test-case on the target.
2. Measure coverage results for a test-case.
3. Restart the target if the target is in a corrupt state.

3.3.1.2.1 Coverage Data Collection Code coverage is usually obtained on source-code level. However for MyFuzzer there is no target source-code available. The solution to this is to go one level deeper and use the concept of code coverage on assembly level. To do this a virtualization layer between the operating system and the target is added. This layer can be instructed to analyze or manipulate the execution of the target. For the purposes of this project it suffices to identify which basic blocks a test-case has executed. A basic block is a sequence of instructions without jumps. So once the first instruction of a basic block is executed all the other instructions will also be executed in order without interruptions. For each test-case it is determined which basic blocks it executed by using pre- and post- signals.

3.3.1.2.2 Data Extraction The raw coverage data is a list of basic blocks. From this two types of coverage information can be deduced. First of all a list of all basic blocks executed can be created together with a count of how many times they have been executed. The second type of coverage information is the transitions between basic blocks. From the raw coverage data it can be determined if a transition between basic block 1 and basic block 2 has occurred before and how many times. These two types of code coverage information are important. For MyFuzzer the task is to get as high as possible coverage.

Using the raw coverage data for test-case t and a hash-table which contains the context it can be deduced how many new basic blocks and how many new transitions where found by test-case t. These two values are the most important coverage metrics. In addition to those two metrics there are several other metrics.

New Basic Blocks Indicate how many new basic blocks have been tested by test-case t.

New Edges Indicates how many new edges between basic blocks have been tested by test-case t.

Test-case Impact Indicates how many basic blocks test-case t has tested. It does not matter how many times each basic block has been tested.

Basic Blocks Executed Less Then x Times Indicates how many basic blocks tested by t where tested less than x times.

Edges Executed Less Then x Times Same as with basic blocks only now for edges.

The last two metrics are divided into four concrete metrics. Less then 10, 100, 1000 and 10000. Note that a basic block is at max only in one of these four.

3.3.2 Dynamic Analysis

To understand the workings of MyFuzzer it is important to understand the interaction between components. MyFuzzer has four important interactions: initialization, execution, evaluation and error recovery. These interactions, together with prerequisites are explained in the following sections.

3.3.2.1 Prerequisites

Before MyFuzzer can be executed the following things have to be in place.

1. The target must be reachable. This means that the fuzzer has to be able to send messages to the target using an ip address and port number.

2. The target monitor must be installed on the target system.

3. The fuzzer needs to know the target communication specification of the target.

4. The target monitor must be reachable.

3.3.2.2 Initialization

Initialization consists of the following steps:

1. Create a configuration file which contains the following information:
    - Target Communication Specification
    - Mutation Rates
    - Code Coverage Weights
2. Run the target monitor on the target system.
3. Run the fuzzer and send a start request to the target monitor.
4. The target monitor will start and monitor the target.

3.3.2.3 Execution

After the initialization phase comes the execution phase. The execution of MyFuzzer works as follows:

- Fuzzer generates a set of test-cases.
- Each test-case is sent to the target in the following fashion:
    1. Fuzzer sends a pre-send request to the target monitor.
    2. Target monitor starts collecting data about the test-case.
    3. Target monitor sends a reply to the fuzzer that it started measuring.
    4. Fuzzer sends the test-case.
    5. Fuzzer sends a post-send request to the target monitor.
    6. Target monitor stop collecting data for the test-case.
    7. Target monitor sends the collected data for the test-case back to the fuzzer.
    8. The fuzzer stores the data for the test-case.
- The fuzzer uses the genetic algorithm to construct the next set of test-cases in the following fashion:
    1. The genetic algorithm scores every test-case based on test-case data.
    2. The genetic algorithm uses the scores to select 50 percent of the test-cases, where test-cases with a higher score have more probability to be selected.

3. The genetic algorithm uses 50-50 cross-over to create new test-cases.
4. The genetic algorithm applies the mutate function to each test-case.
5. The genetic algorithm returns the new generation of test-cases to the fuzzer.

- If the stop criterion is reached the fuzzer stops. Otherwise it will start again at step 2.

3.3.2.4 Evaluation

The goal of the evaluation step is to observer what MyFuzzer did with the target. The most important thing is to find out how many vulnerabilities were found and how much of the target code has been covered. The evaluation step is a combination of automatic and manual methods in order to interpret the data. The utilities used for this are outside the solution scope and are explained in more detail in the experiments section.

3.3.2.4.1 Error Recovery There are several things which might go wrong during the execution phase. First of all the target might crash because it received a bad test-case. Secondly the target might use too much resources so it does not respond. The target might also be in a corrupted state or simply ignore test-cases. In all of these cases the process needs to be restored. This is done by simply restarting the target.

3.4 Implementation

MyFuzzer is based on Sulley, which is an open-source network fuzzer. The fuzzer was written in python. The target monitor was also written in python. This section is not a complete documentation about MyFuzzer it just highlights the most important parts of the implementation. This section will give some details about the two main dependencies of MyFuzzer, DynamoRIO and Sulley. This section also describes how some parts of the genetic algorithm work in MyFuzzer.

3.4.1 Sulley

Sulley [27] is an open-source fuzzer for network applications. MyFuzzer is based on Sulley because Sulley already contained a lot of functionality which could be reused. For example Sulley already implemented the target communication specification and the attack libraries. Sulley also contained the network and process monitor which keep track of the target.

3.4.2 DynamoRIO

In the design section it was explained that a virtualization layer was used for obtaining code coverage information. In the implementation of MyFuzzer this is done using DynamoRIO. DynamoRIO is a run-time manipulation platform which can be applied to executables. DynamoRIO supports clients. A client is a program which is created for a certain task. There is a client called drcov which is used for obtaining coverage information. The only drawback of drcov was that it logged all coverage information to a file and the file was only flushed after the target was finished. So the coverage information became available after the target had finished. This is not useful for MyFuzzer because the coverage data is needed for generating new test-cases. What was needed was that the drcov client would write a coverage file per test-case. This was implemented using signals and nudges, for Linux and Windows respectively.

3.4.2.1 Format of drcov Coverage File

The format of a drcov coverage file is as follows. It starts with some meta information which contains the version and mode of drcov. Then the metadata of the module table is given this states how many modules the target has. Modules are usually the target executable and some libraries (libc). Then some information about the modules are given. This includes module id, path(name), base address, end address and entry address. After the modules comes the basic block table metadata. This states how many basic blocks where executed. After the metadata comes the basic block table itself, which is an ordered list of basic blocks executed. Each basic block is identified by a start address, size in bytes and a module id which indicates which module it belongs to.

3.4.2.2 Linux

DynamoRIO has functionality for defining signal handlers on Linux. The code of the signal handler can be seen in listing 3.1.

Listing 3.1: Signal Handler for Linux

```
static dr_signal_action_t signal_handler(...) {
        ...
    if (info->sig == SIGUSR2) {
        if(drcovlib_dump(NULL) == DRCOVLIB_SUCCESS) {
            dr_fprintf(STDERR, "Succesfully dumped log-file\n");
        }
        return DR_SIGNAL_SUPPRESS;
    }
    return DR_SIGNAL_DELIVER;
}
```

As can be seen if drcov receives a SIGUSR2 signal it will try to dump the current coverage file and continue collecting results in a new file. Files are stored in the following format ”drcov.[processname].[pid].[drcovid].proc”, where drcovid is an eight digit number starting at 00000000, which is incremented for each test-case.

3.4.2.3 Windows

The DynamoRIO implementation on Windows does not support signals but it does support something called nudges. Nudges can be sent using a DynamoRIO utility called drconfig.exe . Basically the same is done as with Linux only now a nudge handler is added. This handler is slightly different but accomplishes the same task. The code can be seen in code snippet 3.2.

Listing 3.2: Nudge Event Handler for Windows

```
static void
event_nudge(void *drcontext, uint64 argument)
{
    int nudge_arg = (int)argument;
    int exit_arg  = (int)(argument >> 32);
    // own code
    if (nudge_arg == 1) {
        // terminate if dump is not succesfull
       if(drcovlib_dump(NULL) != DRCOVLIB_SUCCESS) {
            dr_exit_process(exit_arg);
       }
    }
}
```

This is the only difference between Linux and Windows implementation.

3.4.3 Genetic Algorithm

Like explained in the design section the genetic algorithm has two implementations. However the building blocks for both implementations are the same.

3.4.3.1 Cross-over

There are several possible implementations for the cross-over function. For MyFuzzer it was implemented as 50-50 cross-over with two parents. This means that given two parents the offspring receives a block with 50 percent from one parent and with 50 percent from the other parent. Usually the population of test-cases have all the same structure, but in exceptional cases where this is not the case the offspring contains a block with 50 percent change. Python by default is pass by reference so a deepcopy is used to make sure that parent and child do not influence each other. There is also a possibility of cascading cross-over to lower levels. As explained in the target communication specification section, a message can be built up with blocks and those blocks in turn can contain other blocks. By default cross-over looks at the first level, but if cascade is set to True then it would also perform cross-over on lower levels.

3.4.3.2 Mutators

Mutators are operations which turn an input i into a mutated input i'. MyFuzzer has a different set of mutators per primitive. The most common mutators are swapping the current primitive value with a value from attack library for that primitive, swapping two symbols and random mutation. A complete list of mutators per primitive can be provided upon request. Each mutator has a likelihood associated with it. This likelihood tells how likely it is that this mutator will trigger. Tuning the likelihoods can be a difficult problem.

3.4.3.3 Fitness Function

The fitness function for MyFuzzer is implemented as a function which returns a score between 1 and 100 for each test-case. For the strict implementation of the genetic algorithm the results need to be normalized. This is done by dividing each coverage metric with the max of that metric. That way each metric gets a score between zero and one and the sum of all values of a metrics equals 1. This removes coverage metric bias, for example if metric a would return a score of 50 and metric b returns 1 it might look like a is more important. However they are two different metrics which should have the same influence on the fitness function. That is why the metric values are normalized like this. After that each test-case is scored by multiplying each metric value with its corresponding weight and normalizing to an interval between 1 and 100.

3.4.3.4 Selection

Selection is done by performing a weighted random select on a set of test-cases. The selection step for MyFuzzer always selects 50 percent of the test-cases. So the best test-cases are more likely to be selected.

3.4.4 Fuzzer

As can be seen in figure 3.2 in the design section the fuzzer consists of several components. Probably the most important part of the fuzzer is the test-case generation. This will be explained in the next section.

3.4.4.1 Test-case generation

Test-case generation is the component which generates test-cases according to the given template. The workings of it are very simple. If you have a request consisting of blocks and primitives than a test-case can be generated by simply calling the generate function for that request. The generate function will call the generation function of the blocks and primitives it consists of. Eventually all primitives generate function will be called. It depends on the primitive how data is generated. Test-case generation happens only once. Once the test-cases are generated they are executed, evaluated, selected, mixed and mutated to form a new set of test-cases. The latter are tasks of the genetic algorithm. The primary goal of test-case generation is that it generates a diverse set of test-cases.

3.4.4.1.1 Delimiter The delimiter generation function is simple. With a probability of 0.9 the default value is picked. With the remaining probability of 0.1 a random value from the attack library for delimiters is picked.

3.4.4.1.2 Group With a probability of 0.0001 the value will be empty. Otherwise a random command from the group is picked.

3.4.4.1.3 Random Random generates a random ascii string between the minimum length and maximum length specified.

3.4.4.1.4 Static The static primitive always shows the same default value and does not have functionality for mutation or generation.

3.4.4.1.5 String With a probability of 0.4 the default value is picked. In other cases a random value of the attack library for strings is picked.

3.4.4.1.6 Bitfield With a probability of 0.5 the default value is picked. In other cases a random value of the attack library for bitfields is picked.

3.4.5 Monitors

There are several monitors which monitor target behavior. These monitors are described in the previous section. The network monitor was left unchanged since it did its work properly, the process monitor was changed to add support for multiple targets but the changes are not worth mentioning here. One monitor, the coverage or DynamoRIO monitor, is entirely new and it must be explained how this monitor works.

3.4.5.1 Coverage Monitor

First of all this monitor assumes that the target is running in DynamoRIO. So if we have an executable target t it is assumed that it is not started directly but started using the drrun executable. This assumption is not checked so the person who executes the test must make sure that this assumption holds. The coverage monitor is included as an object in the process monitor and has also a pre-send and post-send function which are executed before and after a test-case respectively. The pre-send function can be seen in code snippet 3.3. It only receives a test-case index, which is used as an identifier to find the corresponding coverage information.

Listing 3.3: Pre-send implementation for the Coverage Monitor

```
def pre_send(self, global_test_number):
        self.current_test = global_test_number
        return True
```

The post send function can be seen in code snippet 3.4.

Listing 3.4: Post-send implementation for the Coverage Monitor

```
def post_send(self):
        self.force_coverage_log()
        self.parse_results()
        if self.feedback == "random":
            self.coverage_metrics = ...
        elif self.feedback == "none":
            self.coverage_metrics = ...
        return self.coverage_metrics
```

First of all the post-send enforces that the coverage file is flushed to disk and creates a new coverage file for the next test-case. After that it parses the results from file and finally it returns the results. If the mode is random or none it will not return the real results. The force_coverage_log function is different for Linux and Windows because the signal handlers of DynamoRIO are implemented differently on Linux and Windows 3.4.2.

3.4.6 Strategies

MyFuzzer has four fuzzing strategies. The strategy has to do with how the genetic algorithm is implemented and the scope of fuzzing. First of all there are two distinct genetic algorithm implementations. The first is called 'strict' and the second 'queue based'. A strict implementation follows the description from the literature in that all basic elements of a genetic algorithm are there. So in a strict implementation there are populations of test-cases, a fitness function and cross-over. In the queue based implementation there is no such thing as a population. There is only a queue and if a test-case is good enough it gets reappended to the queue. Queue based algorithms have the advantage of being fast and require relative little effort to implement. The main reason for including a queue based genetic algorithm is that AFL [16] uses a similar implementation.

The scope has to do with the difference between paths/sessions and nodes/messages. Consider an FTP application where there are two types of messages, "authenticate" and "send file", which can be used to communicate with the FTP server. A node or message is than an instance of one of these types of messages. For example "authenticate user pass" would be a node. A path can be considered a chain of messages. For a given graph, where each node represents a type of message, a node fuzzer tests each node individually whilst a path fuzzer tests each unique path through the graph.

These two parameters lead to the following four strategies of MyFuzzer.

Strategy one: Full GA on Node Level Sulley by default fuzzes per node. This means that once it is done with one node it resets it to the default value and continues with the next node in line. In strategy one this approach is kept. Only now for every node a set of values, the initial population, is generated, partly random and partly statical. This population is executed and each individual in the population is scored based on the coverage metrics explained before and abnormalities detected. Then, like in a real genetic algorithm, a selection is made, cross-over is performed and the individuals are mutated in order to generate the next generation. This is done until a stop criterion is met. The question is that it might not be necessary to go through all the trouble just to emulate a GA perfectly. It might be more efficient to use some elements and leave out some like cross-over and populations. To make use of this strategy, Sulley primitives and blocks have to be rewritten to have more than one possible value. Also a set of mutators have to be selected.

Strategy two: Partial GA on Node Level Like explained in strategy one following the GA approach hundred percent might not be efficient. Therefore in this strategy the AFL method is followed. In this method a queue is created which contain test-cases which have to be executed. If a test-case has a high enough score it is mutated and added to the queue again. Like strategy one this is applied on node level and is stopped until a stop criterion is met. In this case the stop criterion might be that the queue is empty. Cross-over can still be possible but requires some extra attention. For example when a lot of test-cases don't meet the criteria to be added again you might generate some extra test-cases by performing cross-over on them.

Strategy three: Full GA on Global Level Idem to strategy one with the exception that the fuzzing is performed only once on a population of sequences of requests, each sequence representing a run through the target. This has several difficulties. First of all at the initial generation each pass through the program has to be included, in short the population has to be representative. The second difficulty is that cross-over is a lot harder since the difference between individuals is bigger. Another difficulty might be that a new level of mutators has to be added. Mechanisms for request replication might be needed. This method might get very chaotic. On the other hand it tackles the problem of Sulley that it cannot detect multi-request vulnerabilities. This strategy is also done using a full genetic algorithm.

Strategy four: Partial GA on Global Level Uses a queue based implementation of a genetic algorithm. Strategy four is a path fuzzer.

3.5 Argumentation

The solution presented in this chapter matches the requirements given. MyFuzzer uses the rationale described in the introduction chapter 1.3. The way code coverage information is obtained might not be the best since there are other tools like PIN and QEMU, which in all probability could do the same as DynamoRIO. Regardless of efficiency, DynamoRIO is a practical choice. Without much setup coverage information is obtained. Whilst the implementation is on some parts still experimental it can match itself with other fuzzers and fuzzer ideas. In contrast to many other proposed fuzzers, MyFuzzer does not need an initialization step and works right out of the box.

Chapter 4

Experiments

Experiments were done in order to answer the research questions presented in the introduction 1. To reiterate, the research questions are:

- Can the accuracy of a gray-box fuzzer be increased by using a genetic algorithm optimizing for runtime code coverage.

    1. How does MyFuzzer compare to existing fuzzers in terms of finding vulnerabilities and code coverage.
    2. How does the choice of metrics for the fitness function influence the accuracy of MyFuzzer.
    3. How does the choice of fuzzing strategy influence MyFuzzer in terms of finding vulnerabilities and code coverage.

One of the main goals of the research presented in this chapter is to provide evidence that a fuzzer using a genetic algorithm peforms better than a fuzzer without a genetic algorithm. The genetic algorithm for this research is driven by code coverage metrics on assembly level. The second main goal of this research is to provide insight about which fitness metrics are more important for the accuracy of MyFuzzer. A secondary goal arose because of doubt of how to implement MyFuzzer. Because of this multiple fuzzing strategies 3.4.6 were implemented. The secondary goal is to provide evidence about which strategy is the best.

The research consists of three experiments and each experiment consists of multiple executions. Each experiment tries to answer a part of the research question. The remainder of this chapter is divided as follows. First the experimental setup is explained. This section contains information about the fuzzers, targets and systems used for the experiments. After that it is explained how the experiments are built up. The results section shows the most important results of experiment 1, 2 and 3. The analysis section analyzes the results per target per execution per experiment.

Each experiment has a conclusion taking into account all executions and targets. A discussion of the results will be given in the next chapter 5.

4.1 Experimental Setup

Three experiments are executed. Each experiment has a set of fuzzers and a set of targets. Each fuzzer can be configured in multiple ways. Before explaining how the experiments are built up 4.2 it is a good idea to give some information about the targets and fuzzers used in the experiments. This section will also give information about the data collection and measurements.

4.1.1 System Configuration

The experiments are run on the same environment to obtain similar results. The fuzzers are run on a fedora x64 machine using a i5-4690 CPU. The targets are run on an Ubuntu virtual machine using Virtualbox. Some targets are executed on a Windows 7 virtual machine. The target systems have Python installed, together with relevant dependencies to get the monitors running. The target systems also have DynamoRIO 6 available. Multiple virtual machines are used because it allows running several fuzzer instances simultaneously. The virtual machines are clones of each-other to prevent differences between virtual machines.

4.1.2 Targets

The set of targets consists of four targets which can be divided over two categories. The first category can be called 'artificial'. The targets in that category are small example programs which have no real practical use. They however have the benefit that they can be fuzzed in a reasonable amount of time. The second category can be called 'practical'. This category consists of two real FTP server applications, one for Windows and one for Linux. Fuzzing these targets takes more time. The basic idea is to apply a fuzzer first to the artificial targets, take some conclusions from the results and validate it by running the same fuzzer on the practical targets. The set of all targets used for this project can be seen in table 4.1. All targets will be explained in more detail in the following subsections.

4.1.2.1 Vuln_server 1

Vuln_server 1 is a simple TCP application, accepting requests in the form of *[COMMAND][DELIM][DATA]*, where *COMMAND* $\in$ {"COMMAND1", "COMMAND2", "COM-

Target	Unique Vulnerabilities	Operating System	Programming Language
Vuln_server 1	2	Linux	C
Vuln_server 2	3	Linux	C
Proftpd 1.3.3a	At least one	Linux	C
EasyFTP 1.7.0.11	Several	Windows	Unknown

Table 4.1: All Targets used for this Project

MAND3", "REVERSE"}, *DELIM* is a space character and *DATA* is a string. "COMMAND1", "COMMAND2" and "COMMAND3" do nothing. The "REVERSE" command sends *DATA* back in reversed form, so *"REVERSE aab"* will return *"baa"*.

The "REVERSE" command has two vulnerabilities. A buffer overflow which can be exploited if the *DATA* field is too long. For example the string *"REVERSE aaaa...aaa"* will crash the program if the number of a's is large enough. The second vulnerability is a format string vulnerability which can be exploited with strings like *"REVERSE %s%s%s%s%s"*. Vuln_server is a single state program and only accepts one type of message. The target communication specification which MyFuzzer uses for this target can be seen in listing 4.1.

Listing 4.1: Target Communication Specification for Vuln_server 1

```
s_initialize('req1', ...)
s_group(..., ["COMMAND1", "COMMAND2", "COMMAND3", "REVERSE"], ...)
s_delim(' ', ...)
s_string('fuzz', ...)
```

It can be seen that there is one message consisting of three fields. All three fields are non-static and can be mutated with the mutators available for that data-type.

4.1.2.2 Vuln_server 2

Vuln_server 2 is another TCP application and is basically an extension of Vuln_server 1. This application has four internal states. In state 1 the application accepts messages in the form of *[COMMAND]*, where *COMMAND* $\in$ {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L"}. State 2 is reached from state 1 if the message *"B"* was send in state 1. State 2 accepts messages in the form of *[COMMAND][DELIM][DATA]*, where *COMMAND* $\in$ {"COMMAND1", "COMMAND2", "COMMAND3", "REVERSE"}, *DELIM* is a space and *DATA* is a data string. The commands do the same as in Vuln_server 1. State 2 is an endpoint. State 3 is reached from state 1, if in state 1 the message *I* was send. State 3 accepts messages in the form of *[COMMAND][DELIM][DATA]*, where *COMMAND* $\in$ {"A", "B", "C", "D"}, *DELIM* is a space and *DATA* is an integer. State 4 is reached if in state 3 the following message is send *"C 1255"*. In state 4 messages in the form *[COMMAND][DELIM][DATA]* are accepted as valid inputs, where *COMMAND* ∈ {"A", "B", "C", "COPY"}, *DELIM* is a space and *DATA* is a string. The *COPY* command has a buffer overflow vulnerability. The idea behind this target is that the fuzzers now have to get in the right state in order to find vulnerabilities. Since different strategies handle states differently we can expect to see differences between different strategies. Node fuzzers see each state or request as a single search space, so Vuln_server has 4 search spaces. Path fuzzers see each path as a search space, which might cover more than one state. The overview of the protocol can be seen in listing 4.2.

Listing 4.2: Target Communication Specification for Vuln_server 2
```
s_initialize('req1', ...)
s_group('commands1', ["A", "B", "C", "D",
         "E", "F", "G", "H",
         "I", "J", "K", "L"], ...)
s_static('\n')

s_initialize('req2', ...)
s_group('commands2', ["COMMAND1", "COMMAND2",
         "COMMAND3", "REVERSE"], ...)
s_delim(' ', ...)
s_string('fuzz', ...)
s_static('\n')

s_initialize('req3', ...)
s_group('commands3', ["A", "B", "C", "D"], ...)
s_delim(' ',...)
s_bit_field(1255, 32, "bitfield1", format="ascii", ...)
s_static('\n')

s_initialize('req4', ...)
s_group('commands4', ["A", "B", "C", "COPY"], ...)
s_delim(' ', ...)
s_string('fuzz', ...)
s_static('\n')

s.connect(s_get('req1'))
s.connect(s_get('req1'), s_get('req2'),
         dependency = {"commands1" : "B"})
```

```
s.connect(s_get('req1'), s_get('req3'),
        dependency = {"commands1" : "I"})
s.connect(s_get('req3'), s_get('req4'),
        dependency = {"commands3" : "C", "bitfield1" : 1255})
```

The general state overview of Vuln_server 2 can be seen in figure 4.1.

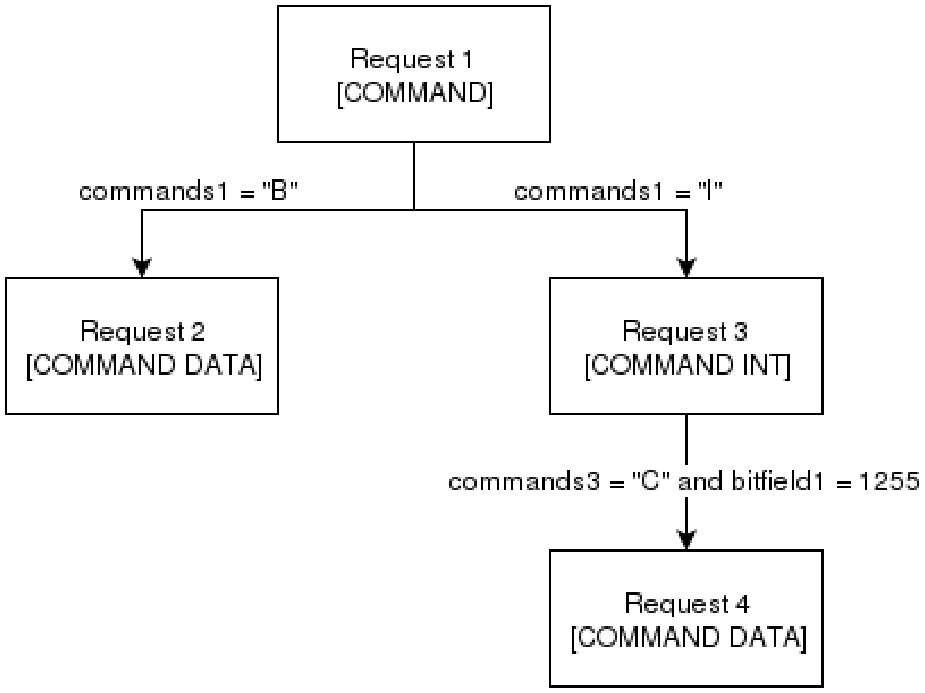


Figure 4.1: Communication Protocol of Vuln_server 2

4.1.2.3 Proftpd 1.3.3a

Proftpd is an FTP server developed for Linux. This version contains at least one vulnerability according to the exploit database [30]. The fuzzer should be able to find this vulnerability. The target communication specification and other settings of Proftpd can be provided upon request. The application itself can be found in the exploit database [30].

4.1.2.4 EasyFTP 1.7.0.11

EasyFTP is a simple FTP server for Windows. EasyFTP has several known vulnerabilities which can be exploited [30]. MyFuzzer should be able to find these vulnerabilities. The application itself can be found in the exploit database [30].

4.1.3 Fuzzers

For the research presented in this report two fuzzers where used, Sulley and MyFuzzer. MyFuzzer has several parameters which where explained in the solution chapter 3. These parameters include fuzzing strategy, metric weights and feedback mode. The first two parameters were explained earlier. The feedback mode is included for research purposes. The normal mode instructs the coverage monitor to return the coverage results for each test-case properly to the fuzzer. The no feedback mode instructs the fuzzer to not send the coverage results back. This has as a consequence that the fuzzer thinks every test-case is equally important. The rationale is that the normal mode should always beat the no feedback mode because it receives proper feedback. The random mode instruct the monitor to return random coverage results. This makes the selection of test-cases unpredictable. Because of these parameters numerous distinct instances of MyFuzzer can be created. For this project we restricted ourselves to the following instances of MyFuzzer.

MyFuzzer strategy 1 A MyFuzzer instance using fuzzing strategy 1 and running in the normal mode.

MyFuzzer strategy 2 A MyFuzzer instance using fuzzing strategy 2 and running in the normal mode.

MyFuzzer strategy 3 A MyFuzzer instance using fuzzing strategy 3 and running in the normal mode.

MyFuzzer strategy 4 A MyFuzzer instance using fuzzing strategy 4 and running in the normal mode.

MyFuzzer strategy 1 - no feedback A MyFuzzer instance using fuzzing strategy 1 and running in the no feedback mode.

MyFuzzer strategy 2 - no feedback A MyFuzzer instance using fuzzing strategy 2 and running in the no feedback mode.

MyFuzzer strategy 3 - no feedback A MyFuzzer instance using fuzzing strategy 3 and running in the no feedback mode.

MyFuzzer strategy 4 - no feedback A MyFuzzer instance using fuzzing strategy 4 and running in the no feedback mode.

MyFuzzer strategy 1 - random feedback A MyFuzzer instance using fuzzing strategy 1 and running in the random mode.

MyFuzzer strategy 2 - random feedback A MyFuzzer instance using fuzzing strategy 2 and running in the random mode.

MyFuzzer strategy 3 - random feedback A MyFuzzer instance using fuzzing strategy 3 and running in the random mode.

MyFuzzer strategy 4 - random feedback A MyFuzzer instance using fuzzing strategy 4 and running in the random mode.

MyFuzzer strategy x - full basic block A MyFuzzer instance using a yet unknown fuzzing strategy and running in normal mode. This instance uses a fitness function which focuses on finding new basic blocks.

MyFuzzer strategy x - full edge A MyFuzzer instance using a yet unknown fuzzing strategy and running in normal mode. This instance uses a fitness function which focuses on finding new edges.

MyFuzzer strategy x - half basic block, edge A MyFuzzer instance using a yet unknown fuzzing strategy and running in normal mode. This instance uses a fitness function which focuses on both finding new edges and new basic blocks.

MyFuzzer strategy x - third basic block, edge, size A MyFuzzer instance using a yet unknown fuzzing strategy and running in normal mode. This instance uses a fitness function which focuses on finding new basic blocks, new edges and test-cases with the largest impact on the target.

The x in the last four instances stands for the best strategy from the first experiment. The words full, half and third indicate the importance of each metric.
The additional parameter values of these instances will be given in the experiment explanation. Most parameter values are constant for each instance of MyFuzzer. Mutation rates however differ per target. Note that throughout this chapter the term fuzzer is applied to either a MyFuzzer instance or Sulley.

4.1.4 Data Collection

Data has to be collected and evaluated after the experiments have been executed. There are three main and two additional data-sources. The main sources are:

Session File The session file is stored on the host machine and is a dictionary of useful information. It stores all crashing test-cases, the coverage per test-case and other useful information.

Crashbin File This file is stored on the target machine and contains a list of all crashes. It can be compared to the session file in order to verify integrity.

Coverage File The coverage file is stored on the target machine and contains coverage information. Every 30 seconds an entry is added to this file giving the current basic block and edge coverage. This file is used for generating the plot showing the total coverage against the number of test-cases.

The secondary sources are:

pcap Files Each test-case sent to the target is stored in a pcap file. Currently these files are not used, but they could be useful in the future since they store the actual data transmitted to the target.

drcov Files Every test-case has a coverage file associated with it. These files describe in detail which basic blocks a test-case encountered. In practice it is too much data and is therefore ignored.

After execution the data integrity is verified using SHA1 hashes. When plotting data, checks between different sources are made in order to further verify the integrity of the results.

4.1.5 Measurements

Measurements help in quantifying the results. The main metrics which will be used for measuring are the number of vulnerabilities found, the number of unique vulnerabilities found, total edge coverage and total basic block coverage. These metrics might be supplemented with times. The first metric indicates how well the GA works, a good GA is expected to find more vulnerabilities (might be duplicates). The second metric evaluates how well a GA can find distinct vulnerabilities. The third and fourth indicate how much code has been covered. Increasing code coverage is an important goal of the genetic algorithm. The measurement can also be used to observe possible correlation between coverage and the process of finding vulnerabilities.

4.2 Experiments

There are three experiments. Experiment 1 is aimed to partially answer research question 1 and 3 . Experiment 2 is also used to answer research question 1 and consequently is also aimed to validate the results from experiment 1. Experiment 3 is aimed to answer research question 2.

4.2.1 Experiment 1

The targets used for this experiment are:

- Vuln_server 1
- Vuln_server 2

The fuzzers used for this experiment are:

- MyFuzzer strategy 1
- MyFuzzer strategy 2
- MyFuzzer strategy 3
- MyFuzzer strategy 4
- MyFuzzer strategy 1 - no feedback
- MyFuzzer strategy 2 - no feedback
- MyFuzzer strategy 3 - no feedback
- MyFuzzer strategy 4 - no feedback
- MyFuzzer strategy 1 - random feedback
- MyFuzzer strategy 2 - random feedback
- MyFuzzer strategy 3 - random feedback
- MyFuzzer strategy 4 - random feedback
- Sulley

The goal of this experiment is partially answering research question 1 and answering research question 3. The last goal is important since it has influence on the other experiments. The best strategy from this experiment will be used for further experiments and the other strategies will be ignored. The reason for this is that keeping all strategies would be too time consuming. Experiment 1 is executed twice. Each execution differs slightly. The second differs from the first execution in that all targets are run on the same VM and that ASLR is explicitly disabled.

The metric weights3.3.1.2.2 for this experiment are:

New Edges 0.45

New Basic Blocks 0.35

Test-case Impact 0.02

Number of Edges Executed Between 1 and 10 Times 0.06

Number of Edges Executed Between 9 and 100 Times 0.02

Number of Edges Executed Between 99 and 1000 Times 0.015

Number of Edges Executed Between 999 and 10000 Times 0.01

Number of Basic Blocks Executed Between 1 and 10 Times 0.05

Number of Basic Blocks Executed Between 9 and 100 Times 0.01

Number of Basic Blocks Executed Between 99 and 1000 Times 0.01

Number of Basic Blocks Executed Between 999 and 10000 Times 0.005

The main focus is on new basic blocks and new edges found, the rationale behind this is that new pieces of code need to be tested more frequently in order to ensure that it does not contain vulnerabilities.

The mutation rates differ for each target. The mutation rates and target communication specification for each target can be provided upon request.

4.2.2 Experiment 2

The targets used for experiment 2 are:

- Proftpd 1.3.3a
- EasyFTP 1.7.0.11
- Vuln_server 1
- Vuln_server 2

The fuzzers used for this experiment are:

- MyFuzzer strategy x
- MyFuzzer strategy x - no feedback
- MyFuzzer strategy x - random feedback
- Sulley

The goal of this experiment is to answer research question 1. The strategy used in this experiment is dependent on the results from experiment 1. The best strategy from experiment 1 will be used in experiment 2 and 3. The omission of other strategies is because including all strategies would be too time consuming.

The coverage metric weights used in this experiment are the same as in experiment 1.

The mutation rates and target communication specification for each target can be provided upon request.

Experiment 2 is divided over three executions. Execution 2, like in experiment 1, has ASLR explicitly disabled to see if ASLR interferes with MyFuzzer. In the third execution only the target executables are monitored and the target dependencies, like libraries and other resources, are ignored.

4.2.3 Experiment 3

The targets used for this experiment are:

- Vuln_server 1
- Vuln_server 2
- Proftpd 1.3.3a
- EasyFTP 1.7.0.11

The fuzzers used for this experiment are:

- MyFuzzer strategy x - full basic block
- MyFuzzer strategy x - full edge
- MyFuzzer strategy x - half basic block, edge
- MyFuzzer strategy x - third basic block, edge, size

The goal of this experiment is to answer research question 2. The first two targets are executed twice. When appropriate the results of MyFuzzer strategy x from experiment 2 will also be used.

The coverage metric weights for this experiment differ for each fuzzer. For MyFuzzer strategy x the same coverage metric weights as in experiment 1 and 2 are used. For full basic block the following coverage metric weights are used:

New Basic Blocks 0.875

Number of Basic Blocks Executed Between 1 and 10 Times 0.1

Number of Basic Blocks Executed Between 9 and 100 Times 0.02

Number of Basic Blocks Executed Between 99 and 1000 Times 0.004

Number of Basic Blocks Executed Between 999 and 10000 Times 0.001

For full edge the following coverage metric weights are used:

New Edges 0.875

Number of Edges Executed Between 1 and 10 Times 0.1

Number of Edges Executed Between 9 and 100 Times 0.02

Number of Edges Executed Between 99 and 1000 Times 0.004

Number of Edges Executed Between 999 and 10000 Times 0.001

For half basic block, edge the following coverage metric weights are used:

New Basic Blocks 0.4375

Number of Basic Blocks Executed Between 1 and 10 Times 0.05

Number of Basic Blocks Executed Between 9 and 100 Times 0.01

Number of Basic Blocks Executed Between 99 and 1000 Times 0.002

Number of Basic Blocks Executed Between 999 and 10000 Times 0.0005

New Edges 0.4375

Number of Edges Executed Between 1 and 10 Times 0.05

Number of Edges Executed Between 9 and 100 Times 0.01

Number of Edges Executed Between 99 and 1000 Times 0.002

Number of Edges Executed Between 999 and 10000 Times 0.0005

For third basic block, edge, size the following coverage metric weights are used:

New Basic Blocks 0.2625

Number of Basic Blocks Executed Between 1 and 10 Times 0.05

Number of Basic Blocks Executed Between 9 and 100 Times 0.01

Number of Basic Blocks Executed Between 99 and 1000 Times 0.002

Number of Basic Blocks Executed Between 999 and 10000 Times 0.0005

New Edges 0.2625

Number of Edges Executed Between 1 and 10 Times 0.05

Number of Edges Executed Between 9 and 100 Times 0.01

Number of Edges Executed Between 99 and 1000 Times 0.002

Number of Edges Executed Between 999 and 10000 Times 0.0005

Test-case Impact 0.35

The metric weight is 0 when a specific metric is not listed.

The mutation rates can be provided upon request.

This experiment is executed thrice. The first time it is executed on Vuln_server 1 Vuln_server 2 and Proftpd. The second time it is run with ASLR disabled, which could have given some interference. The third time it only takes coverage for the target executable into account and does not include the libraries the targets use.

4.3 Results

The collected data 4.1.4 is used to create graphs and tables which summarize the results. The number of graphs and the size of the table can differ slightly for different targets. In most cases there are two graphs. The first graph displays basic block coverage against the number of test-cases and the second graph displays edge coverage against the number of test-cases. The table contains an entry for each fuzzer. Each entry displays information like the number of test-cases, the number of vulnerabilities found and the number of unique vulnerabilities found. The results presented in this section are referred to in the analysis section 4.4.

Below only the most relevant results are displayed. The other results can be provided upon request.

4.3.1 Results for Experiment 1

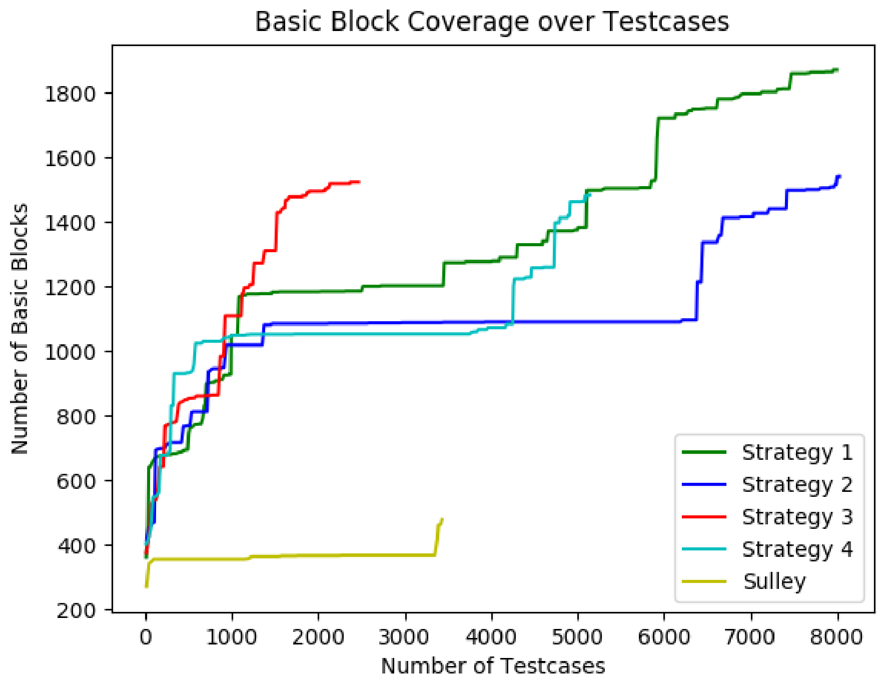
4.3.1.1 Vuln_server 1

The following table shows the results of experiment 1 for Vuln_server 1.

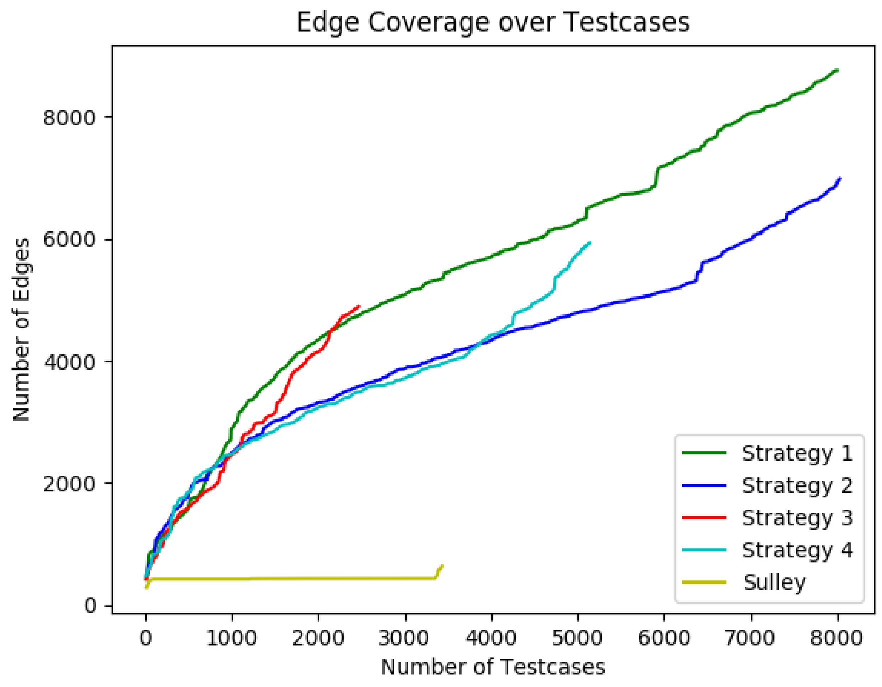
Fuzzer	Test-cases	Vulnerabilities	Unique Vulnerabilities
Sulley	3430	5	1
Strategy 1	8000	409	2
Strategy 2	8031	279	2
Strategy 3	2465	86	2
Strategy 4	5143	274	2
Strategy 1 - no feedback	8000	210	2
Strategy 1 - random feedback	8000	47	2
Strategy 3 - no feedback	2149	80	2
Strategy 3 - random feedback	2790	62	2

Table 4.2: Results for Vuln_server 1 in Experiment 1, Execution 1

The following graphs show the differences between strategies better.



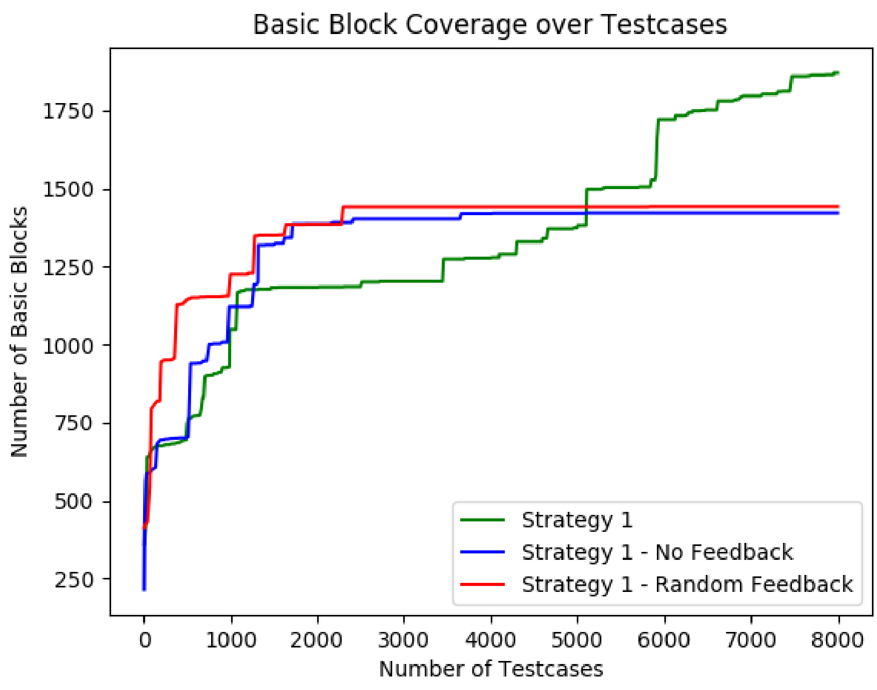
(a) Basic Block Coverage



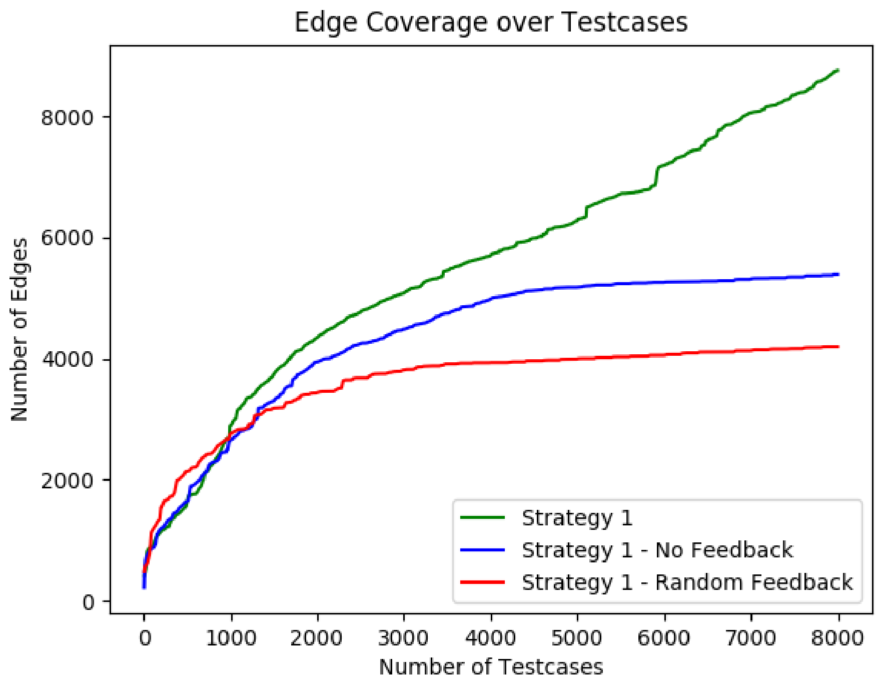
(b) Edge Coverage

Figure 4.2: Results for Vuln_server 1 in Experiment 1, Execution 1 (only Strategies)

The following graphs show the results for strategy 1 for all modes.



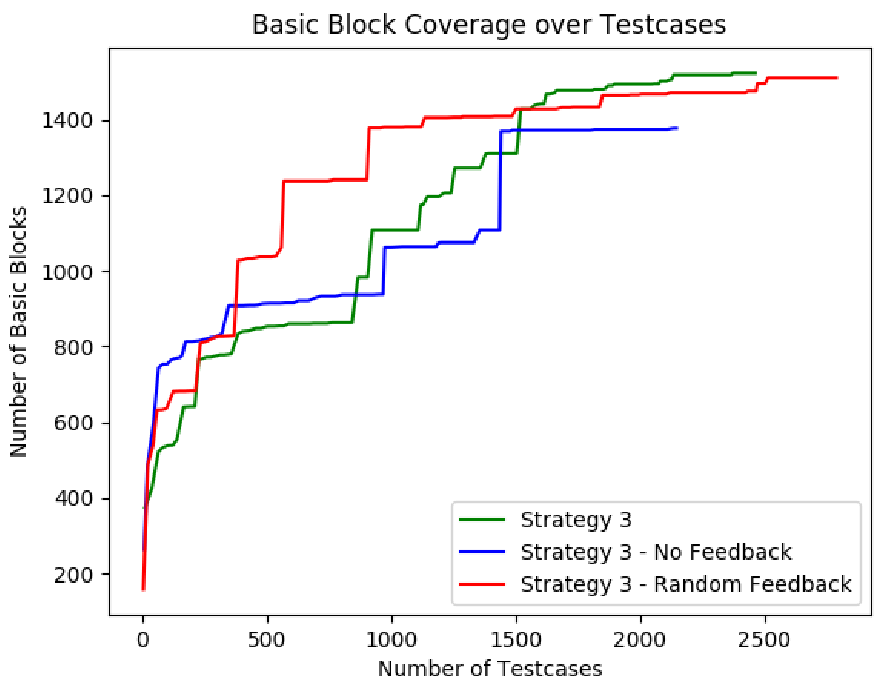
(a) Basic Block Coverage



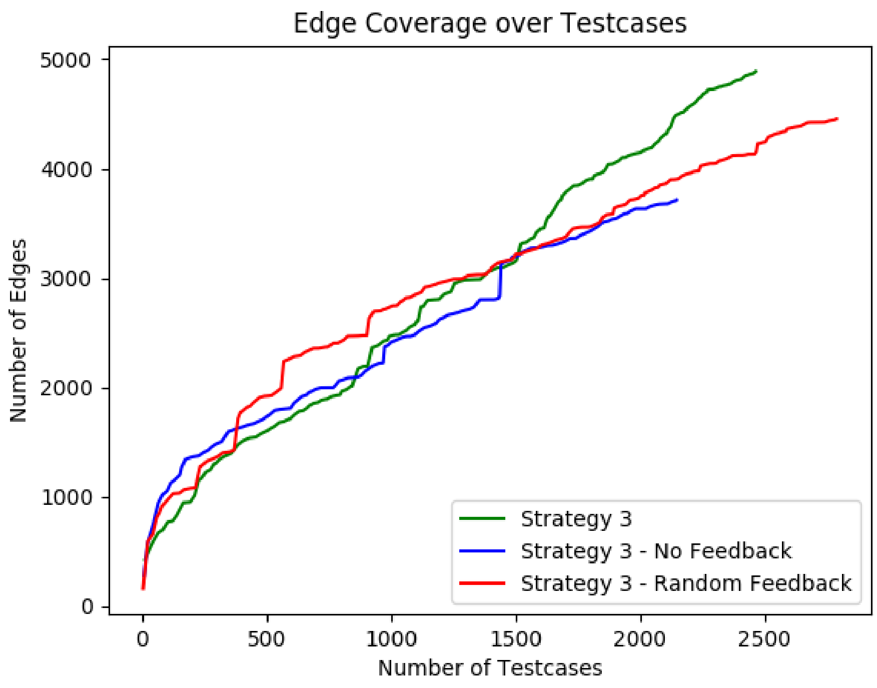
(b) Edge Coverage

Figure 4.3: Results for Vuln_server 1 in Experiment 1, Execution 1 (only Strategy 1)

The following graphs show the results for strategy 3 for all modes.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.4: Results for Vuln_server 1 in Experiment 1, Execution 1 (only Strategy 3)

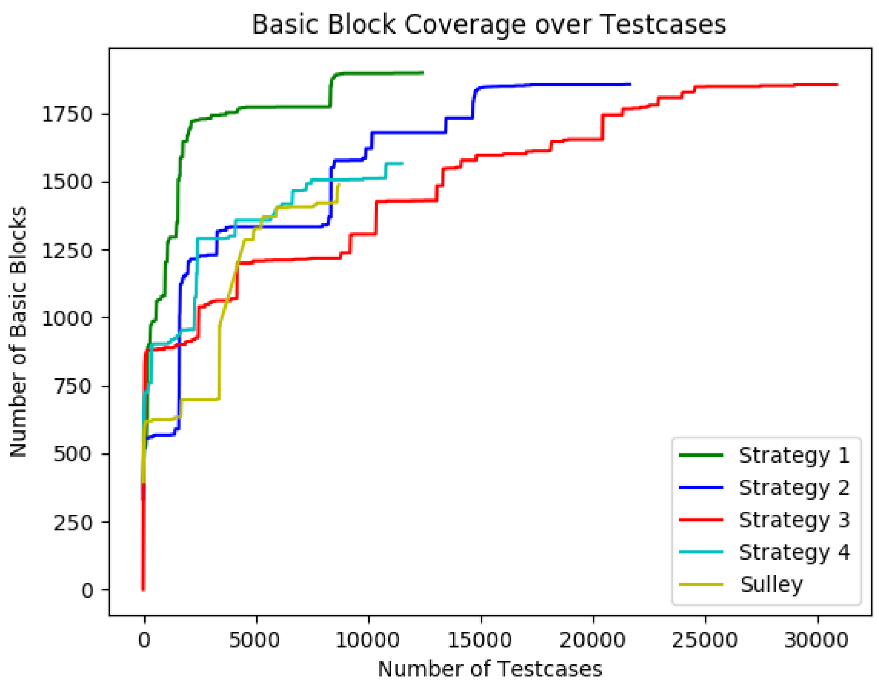
4.3.1.2 Vuln_server 2

The following table shows the results of experiment 1 for Vuln_server 2.

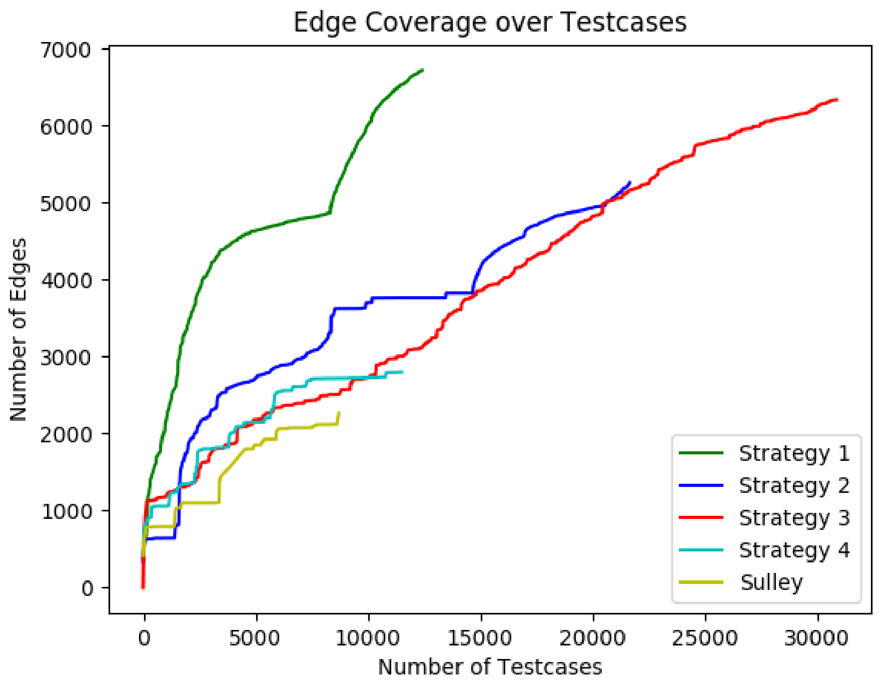
Fuzzer	Test-cases	Vulnerabilities	Unique Vulnerabilities
Sulley	7647	9	2
Strategy 1	12416	248	3
Strategy 2	21643	410	3
Strategy 3	30849	201	3
Strategy 4	11503	10	2
Strategy 1 - no feedback	12416	89	3
Strategy 1 - random feedback	12416	45	3
Strategy 2 - no feedback	19968	230	3
Strategy 2 - random feedback	80000	744	3
Strategy 4 - no feedback	11200	15	3
Strategy 4 - random feedback	16536	13	2

Table 4.3: Results for Vuln_server 2 in Experiment 1, Execution 1

The following graphs show the differences between strategies better.



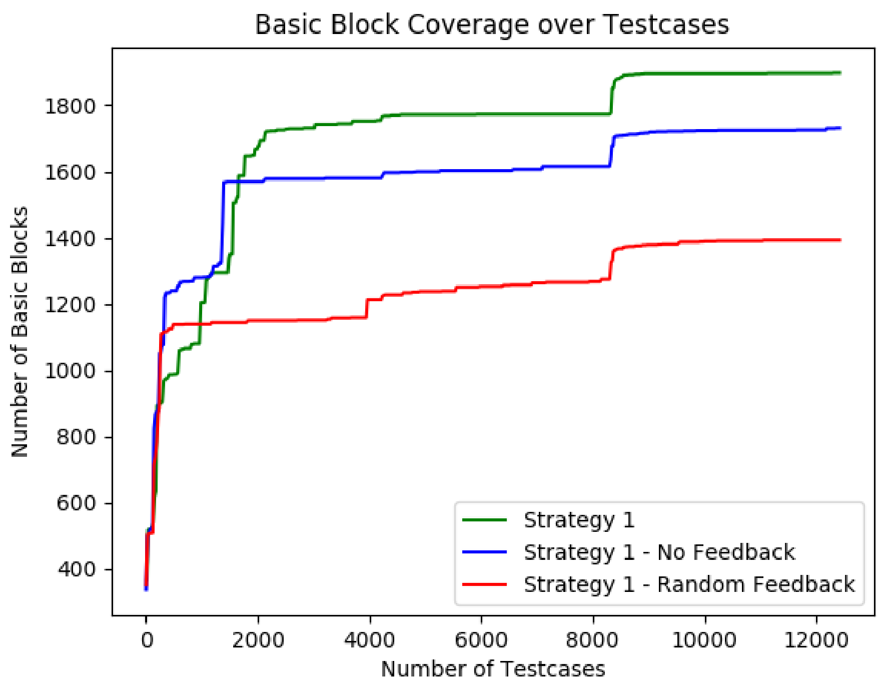
(a) Basic Block Coverage



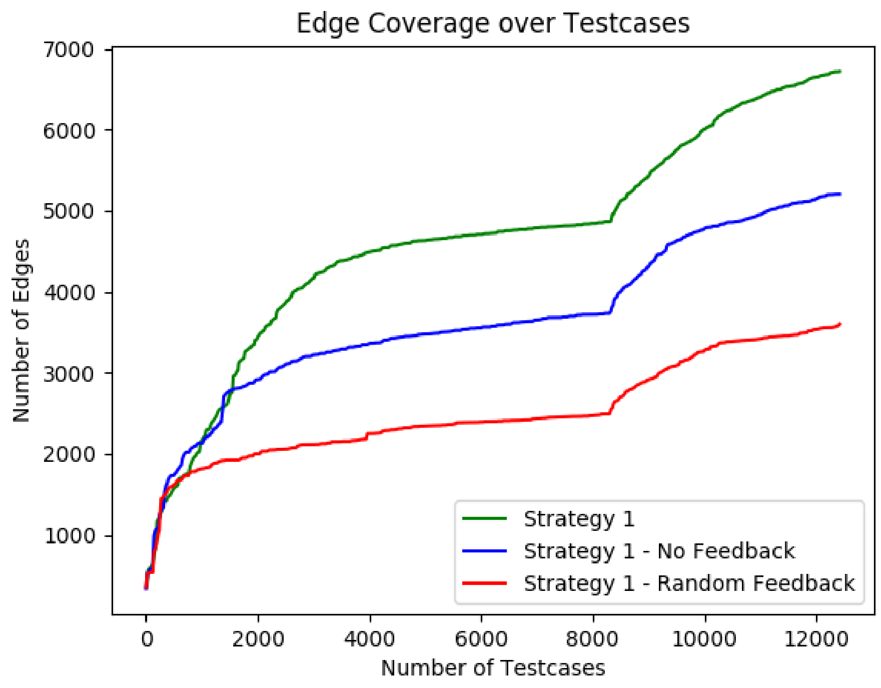
(b) Edge Coverage

Figure 4.5: Results for Vuln_server 2 in Experiment 1, Execution 1 (only Strategies)

The following graphs show the results for strategy 1 for all modes.



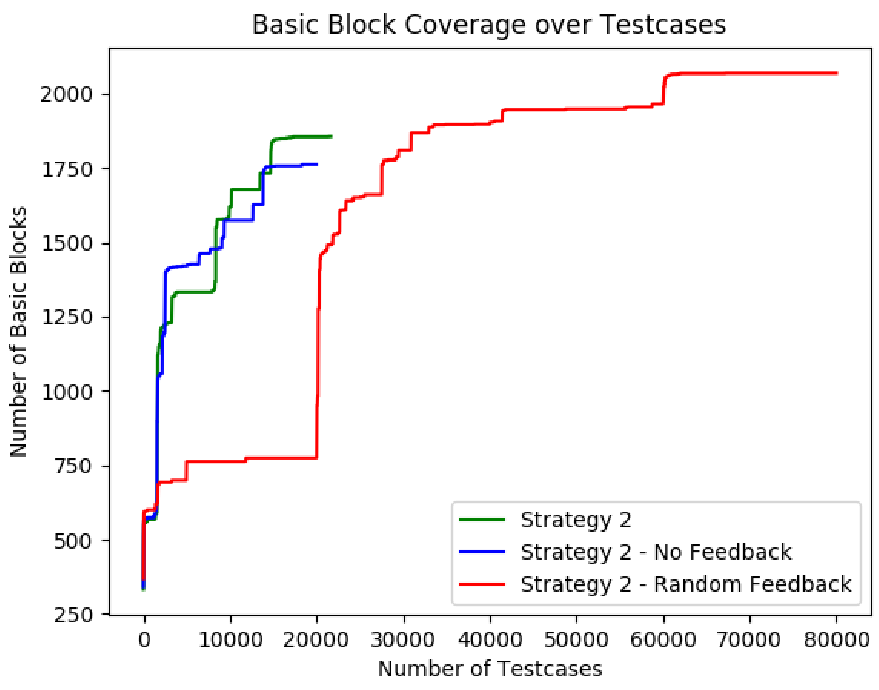
(a) Basic Block Coverage



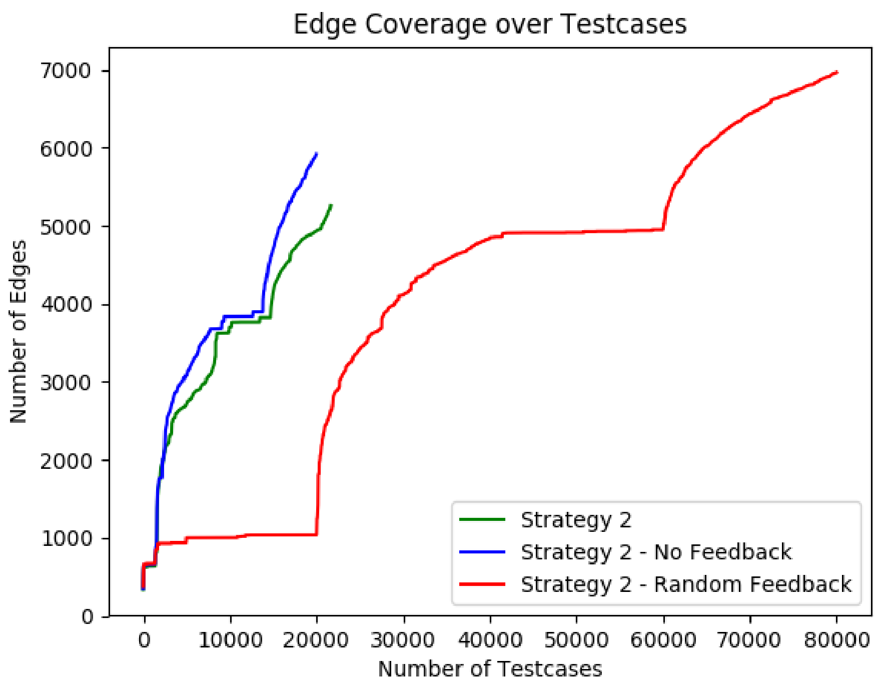
(b) Edge Coverage

Figure 4.6: Results for Vuln_server 2 in Experiment 1, run 1 (only Strategy 1)

The following graphs show the results for strategy 2 for all modes.



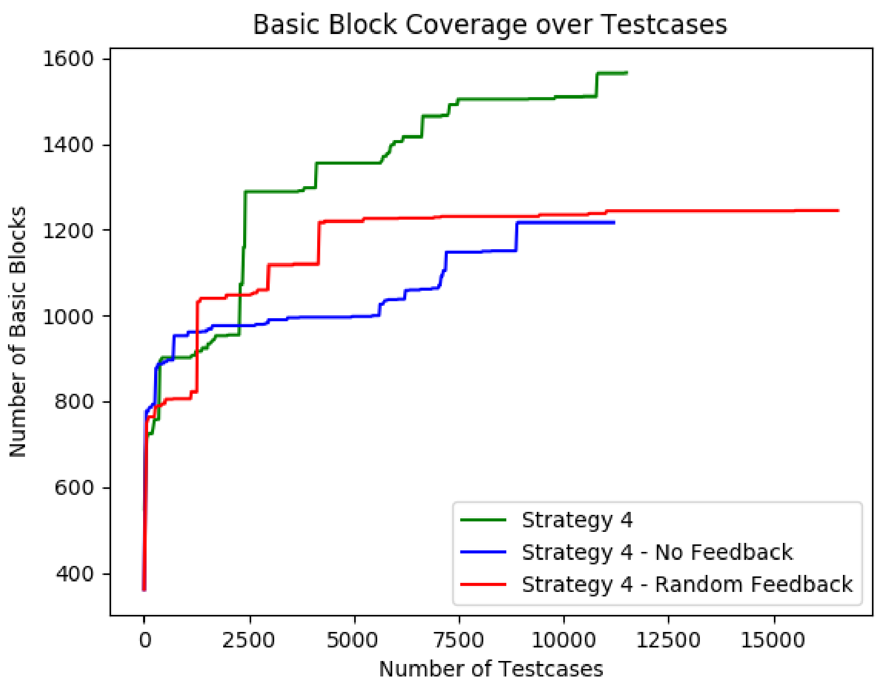
(a) Basic Block Coverage



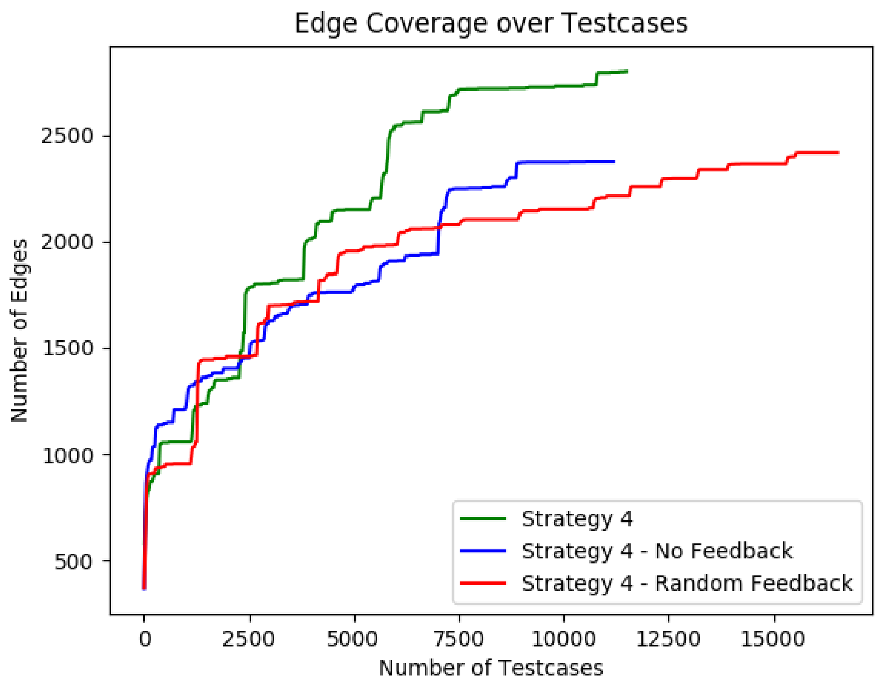
(b) Edge Coverage

Figure 4.7: Results for Vuln_server 2 in Experiment 1, Execution 1 (only Strategy 2)

The following graphs show the results for strategy 4 for all modes.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.8: Results for Vuln_server 2 in Experiment 1, Execution 1 (only Strategy 4)

4.3.2 Results for Experiment 2

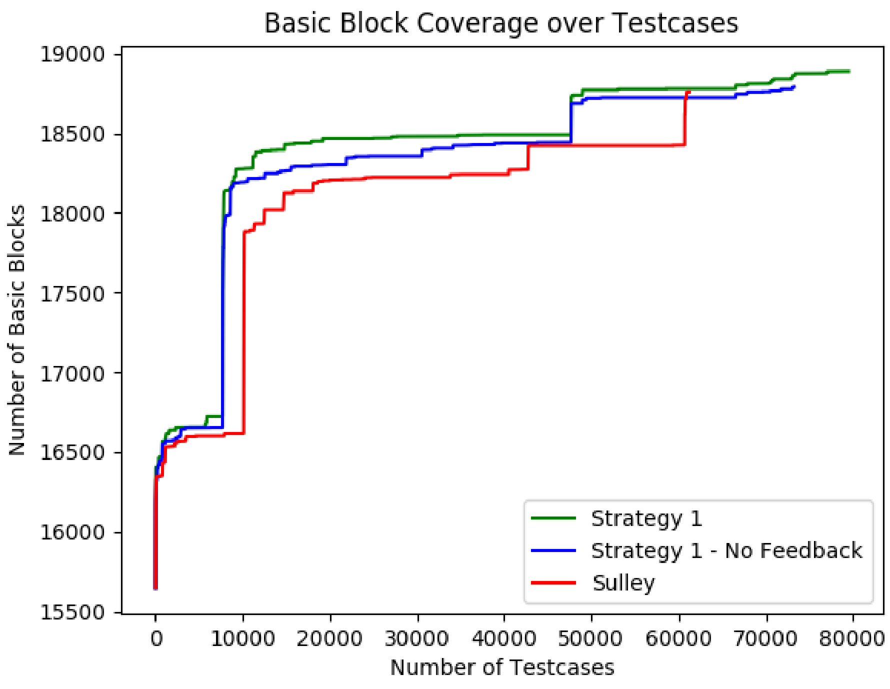
4.3.2.1 EasyFTP

The following table shows the results of experiment 2 for EasyFTP 1.7.0.11 .

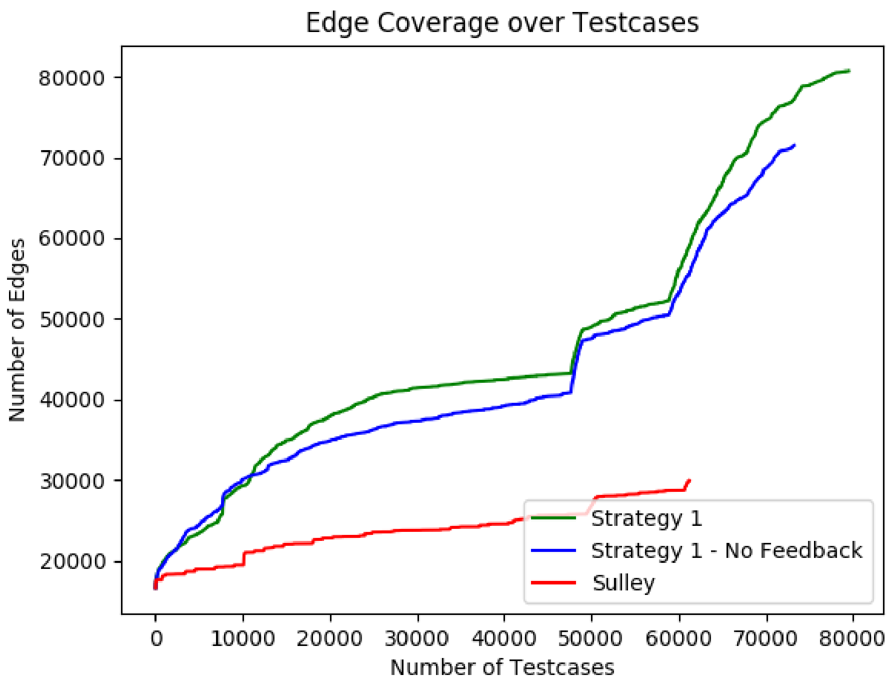
Fuzzer	Test-cases	Vulnerabilities	Unique Vulnerabilities
Sulley	57004	17	7
Strategy 1	79494	53	6
Strategy 1 - no feedback	73264	83	7

Table 4.4: Results for EasyFTP 1.7.0.11 in Experiment 2, Execution 1

The following graphs show the results for EasyFTP of experiment 2 Execution 1, including all fuzzers.



(a) Basic Block Coverage for the Server Executable

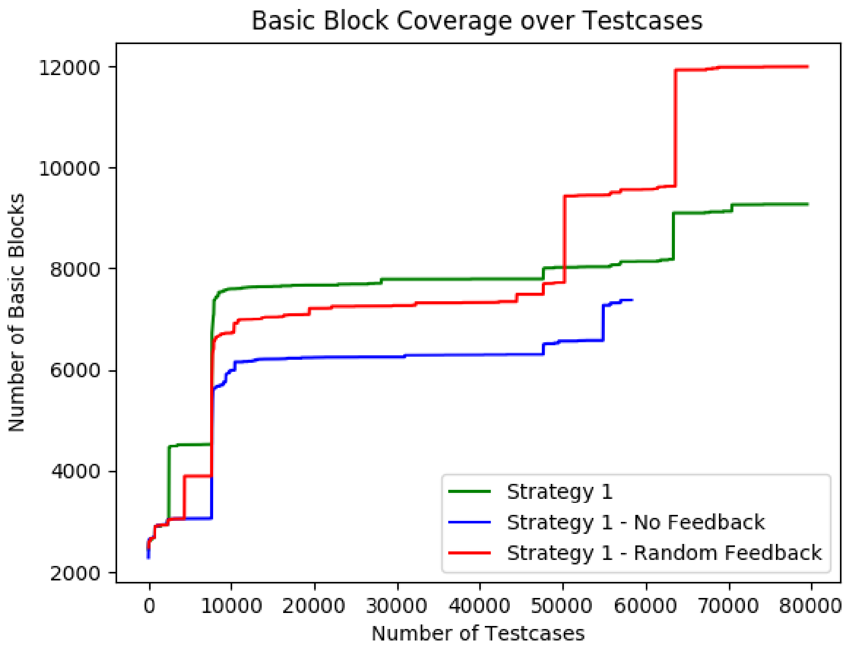


(b) Edge Coverage for the Server Executable

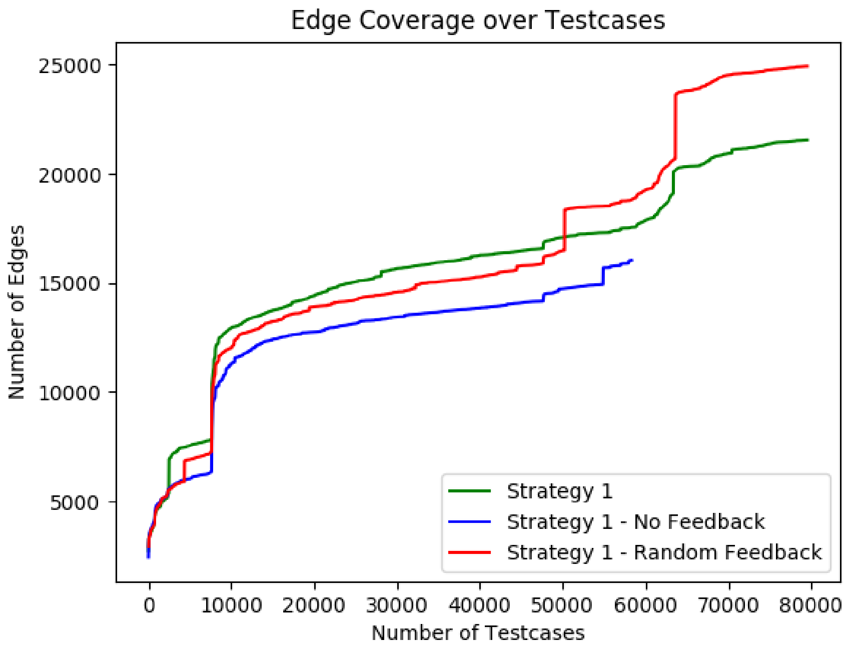
Figure 4.9: Results for EasyFTP 1.7.0.11 in Experiment 2, Execution 1

4.3.2.2 Proftpd

The following graphs show the results for Proftpd1.3.3a of experiment 2 Execution 1, including all fuzzers.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.10: Results for Proftpd 1.3.3a in Experiment 2, Execution 1

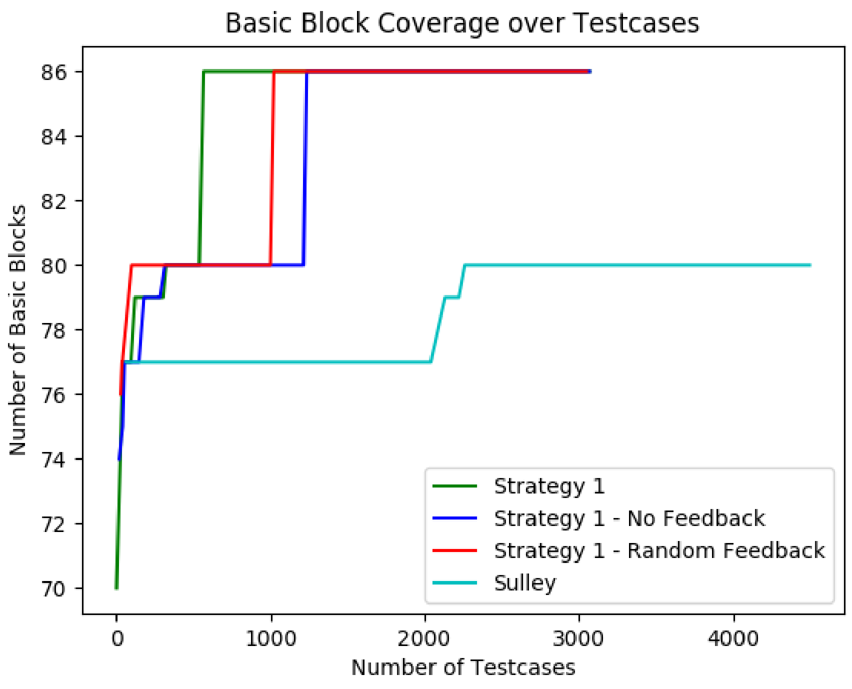
4.3.2.3 Vuln_server 1

The following table shows the results of experiment 2 for Vuln_server 1.

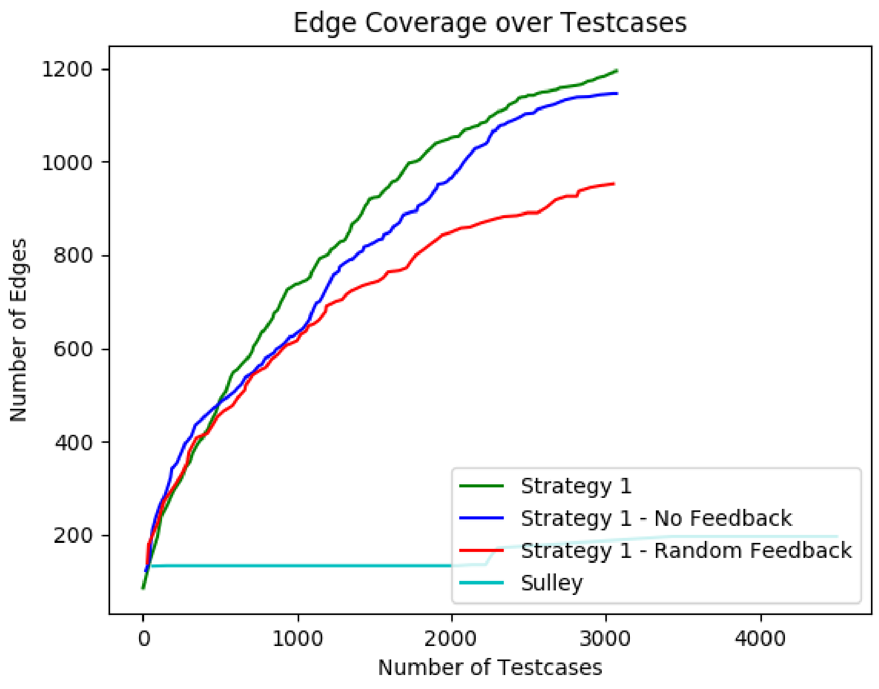
Fuzzer	Test-cases	Vulnerabilities	Unique Vulnerabilities
Sulley	3430	5	1
Strategy 1	3072	145	2
Strategy 1 - no feedback	3072	128	2
Strategy 1 - random feedback	3072	34	2

Table 4.5: Results for Vuln_server 1 in Experiment 2, Execution 3

The following graphs show the results for Vuln_server 1 of experiment 2 Execution 3, including all fuzzers.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.11: Results for Vuln_server 1 in Experiment 2, Execution 3

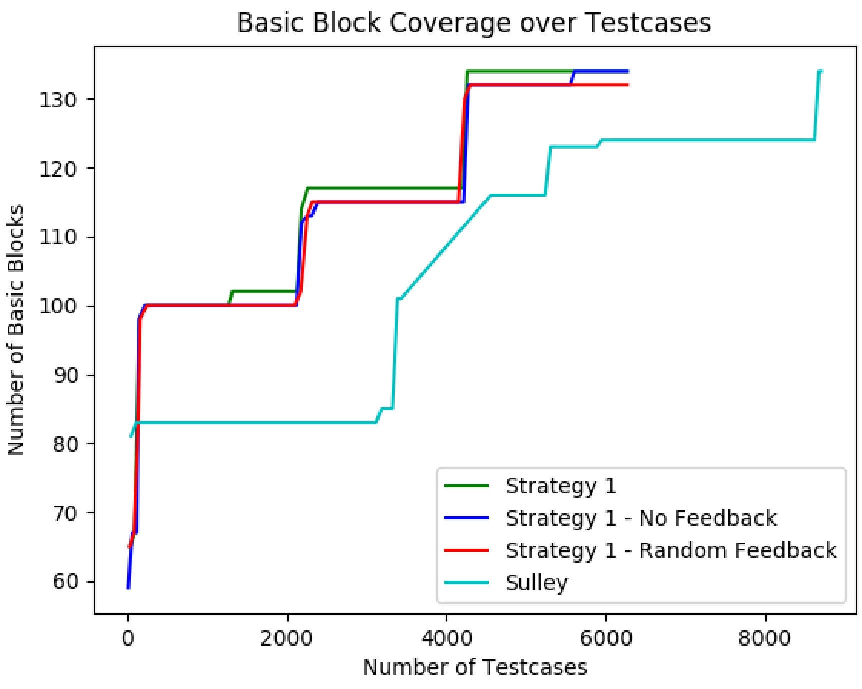
4.3.2.4 Vuln_server 2

The following table shows the results of experiment 2 for Vuln_server 2.

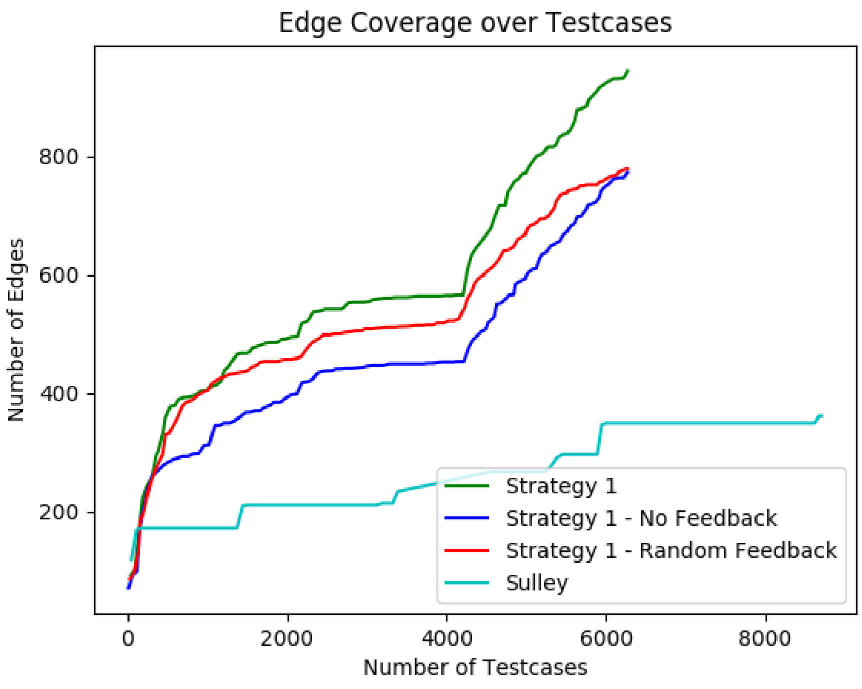
Fuzzer	Test-cases	Vulnerabilities	Unique Vulnerabilities
Sulley	7629	2	2
Strategy 1	6272	59	3
Strategy 1 - no feedback	6272	53	3
Strategy 1 - random feedback	6272	40	3

Table 4.6: Results for Vuln_server 2 in Experiment 2, Run 3

The following graphs show the results for Vuln_server 2 of experiment 2 Execution 3, including all fuzzers.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.12: Results for Vuln_server 2 in Experiment 2, Execution 3

4.3.3 Results for Experiment 3

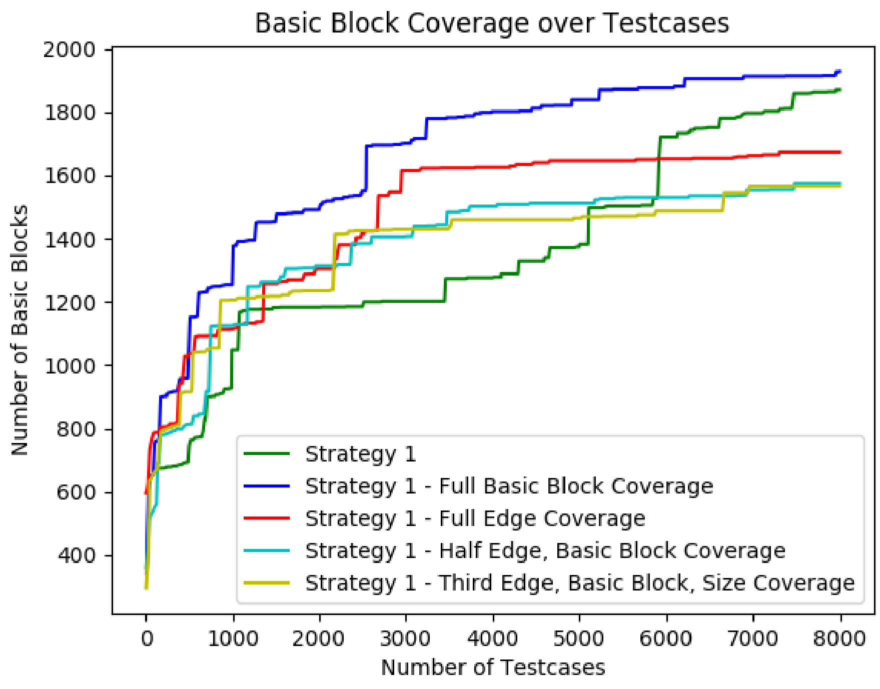
4.3.3.1 Vuln_server 1

The following table shows the results of experiment 3 for Vuln_server 1.

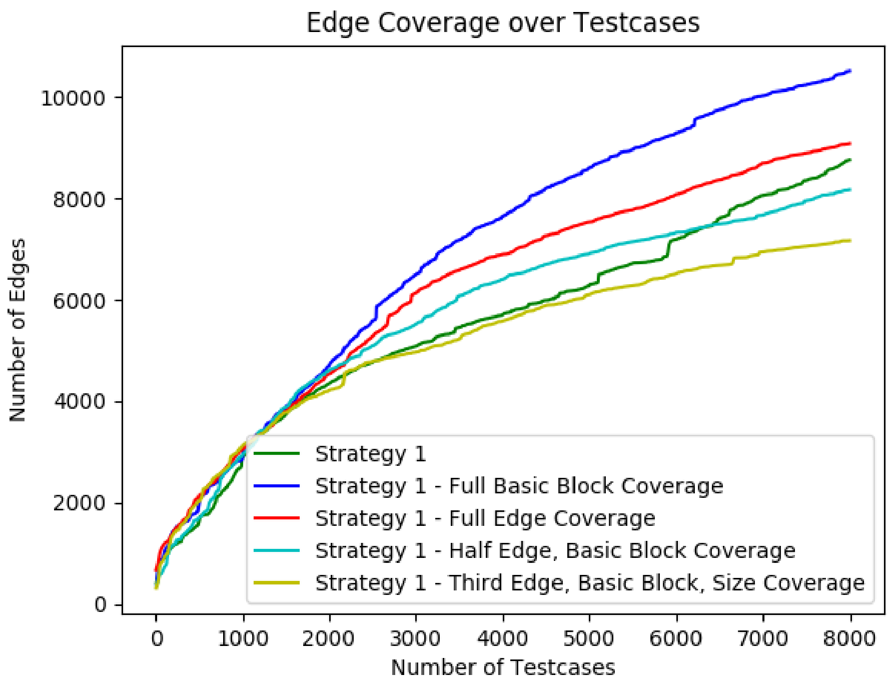
Fuzzer	Test-cases	Vulnerabilities	Unique
Strategy 1	8000	409	2
Strategy 1 - Basic Block	8000	728	2
Strategy 1 - Edge	8000	468	2
Strategy 1 - Edge, Basic Block	8000	359	2
Strategy 1 - Edge, Basic Block, Size	8000	237	2

Table 4.7: Results for Vuln_server 1 in Experiment 3, Execution 1

The following graphs show the results for Vuln_server 1 of experiment 3 Execution 1, including all fuzzers.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.13: Results for Vuln_server 1 in Experiment 3, Execution 1

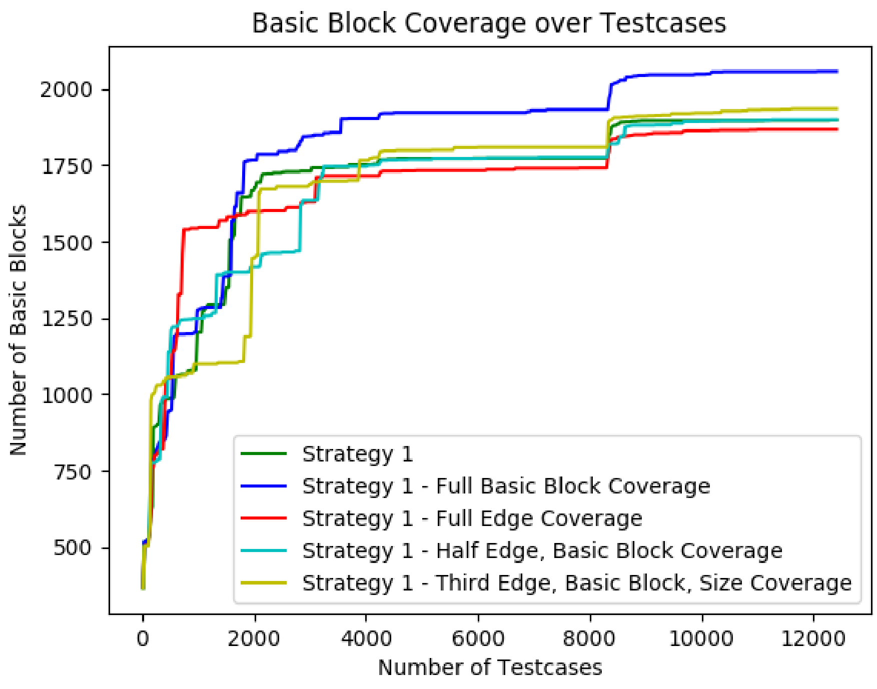
4.3.3.2 Vuln_server 2

The following table shows the results of experiment 3 for Vuln_server 2.

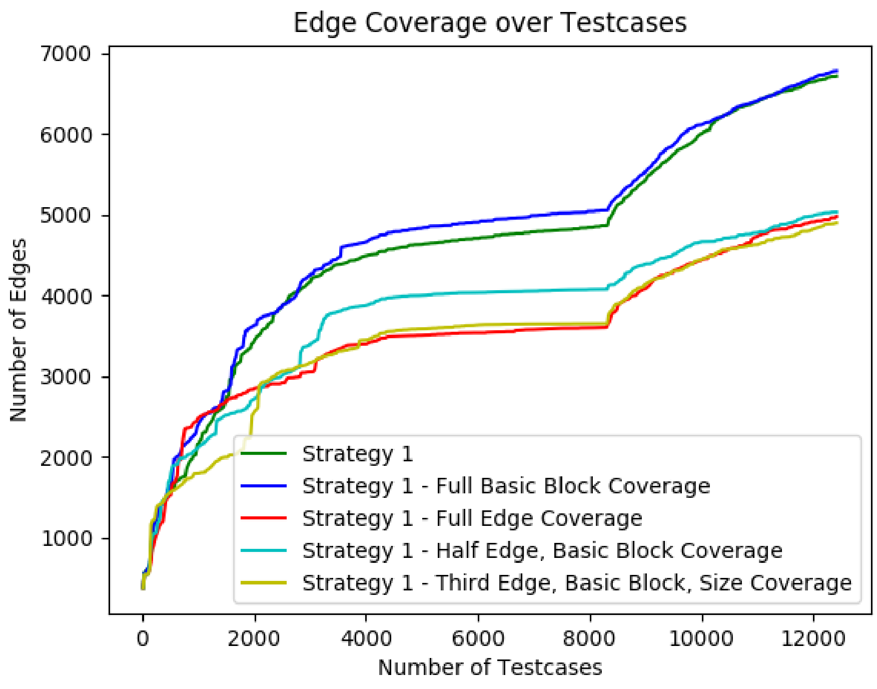
Fuzzer	Test-cases	Vulnerabilities	Unique
Strategy 1	12416	248	3
Strategy 1 - Basic Block	12416	269	3
Strategy 1 - Edge	12416	122	3
Strategy 1 - Edge, Basic Block	12416	145	3
Strategy 1 - Edge, Basic Block, Size	12416	108	3

Table 4.8: Results for Vuln_server 2 in Experiment 3, Execution 1

The following graphs show the results for Vuln_server 2 of experiment 3 Execution 1, including all fuzzers.



(a) Basic Block Coverage

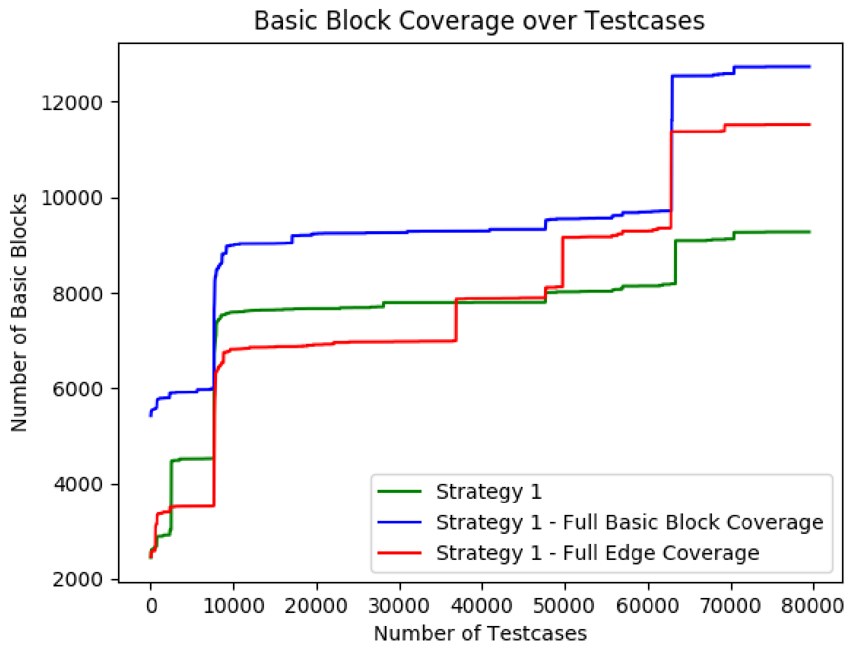


(b) Edge Coverage

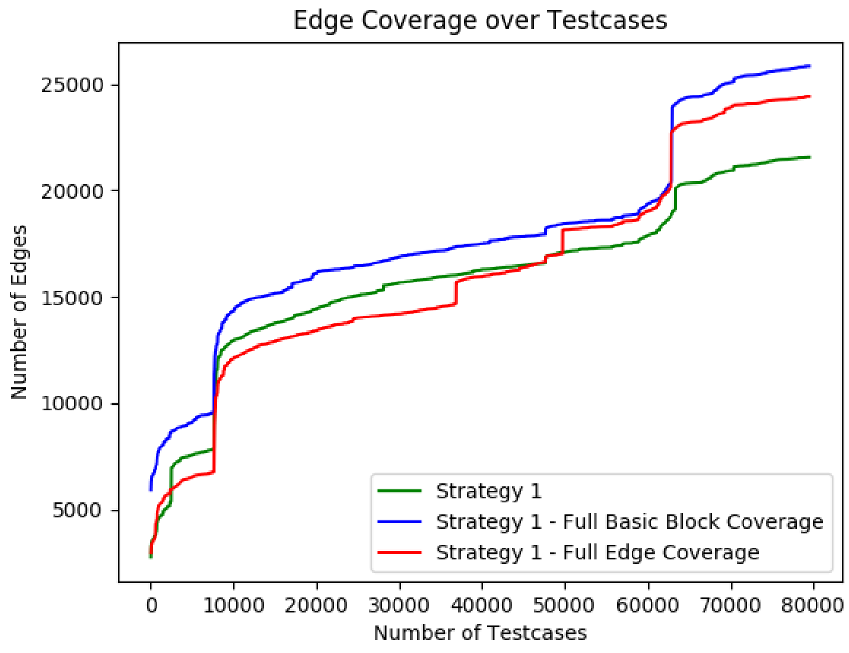
Figure 4.14: Results for Vuln_server 2 in Experiment 3, Execution 1

4.3.3.3 Proftpd

The following graphs show the results for Proftpd1.3.3a of experiment 3 Execution 1, including all fuzzers.



(a) Basic Block Coverage



(b) Edge Coverage

Figure 4.15: Results for Proftpd 1.3.3a in Experiment 3, Execution 1

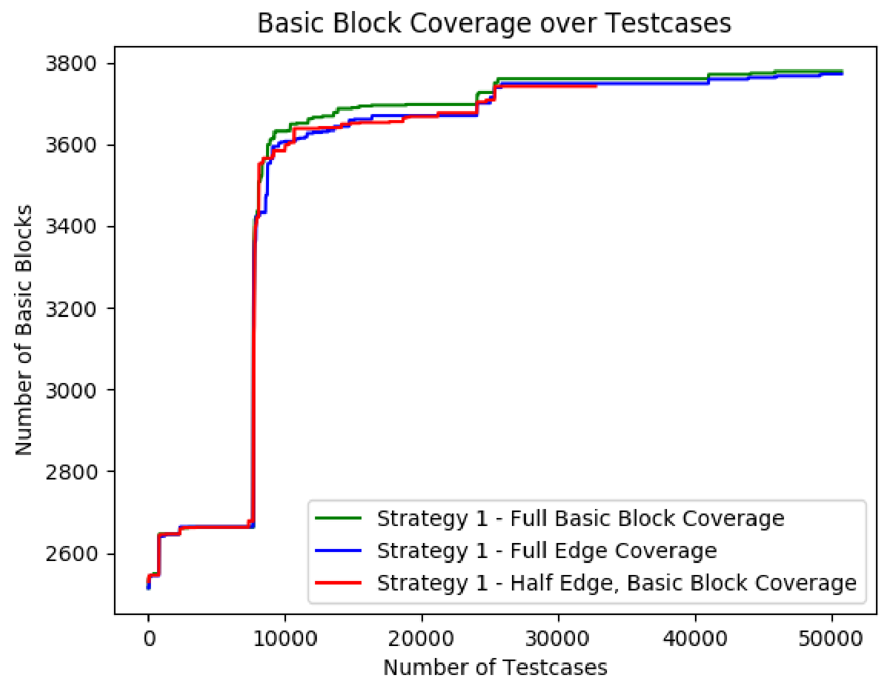
4.3.3.4 EasyFTP

The following table shows the results of experiment 2 for EasyFTP 1.7.0.11 .

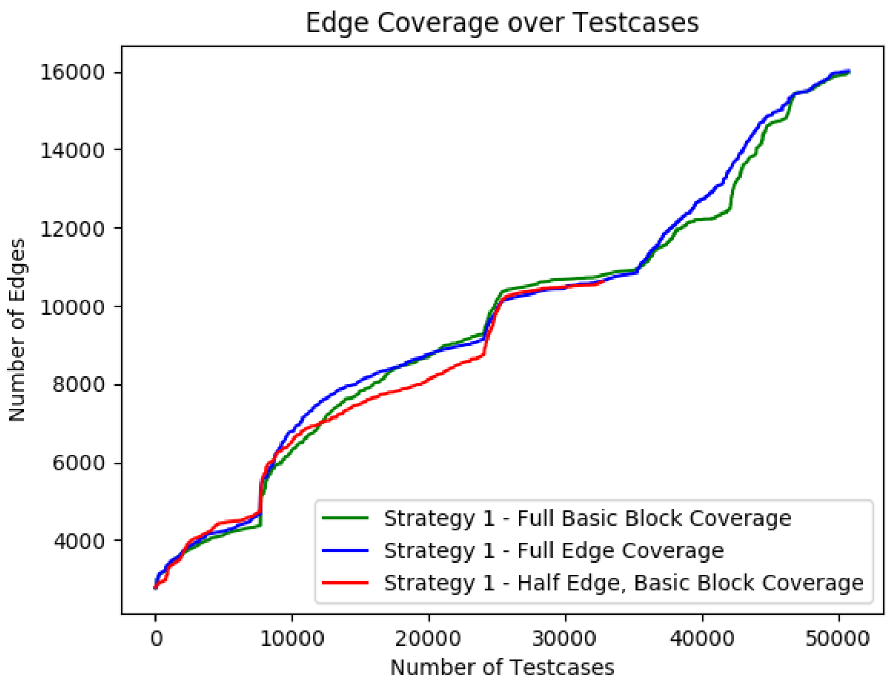
Fuzzer	Test-cases	Vulnerabilities	Unique Vulnerabilities
Strategy 1 - Full Basic Block	50758	134	5
Strategy 1 - Full Edge	50758	50	7
Strategy 1 - Half Edge, Basic Block	32748	7	5

Table 4.9: Results for EasyFTP 1.7.0.11 in Experiment 3, Execution 3

The following graphs show the results for EasyFTP of experiment 3 Execution 3, including all fuzzers.



(a) Basic Block Coverage for the Console Executable



(b) Edge Coverage for the Server Executable

Figure 4.16: Results for EasyFTP 1.7.0.11 in Experiment 3, Execution 3

4.4 Analysis

In this section the results presented in 4.3 are analyzed. For each fuzzer it is analyzed how well it did. The performance of a fuzzer is dependent on two things: the ability to find vulnerabilities and the ability to optimize code coverage. Finding vulnerabilities is evaluated on two main metrics, the number of vulnerabilities found and the number of unique vulnerabilities found. The number of vulnerabilities found might contain duplicates and false positives so it is of minor importance. The number of unique vulnerabilities is the most important metric. Coverage is evaluated using two types of graphs. The first displays edge coverage against the number of test-cases and the second displays basic block coverage against the number of test-cases.

4.4.1 Experiment 1

Experiment 1 has been executed twice on both Vuln_server applications.

4.4.1.1 Vuln_server 1

Vuln_server 1 was executed twice for experiment 1. The results for the first execution can be seen in 4.3.1.1. The results for the second execution can be provided upon request. No significant differences were observed between the different executions, so the impact of ASLR is minimal.

It can be seen in table 4.2 that Sulley performs worse than other fuzzers in finding vulnerabilities. All other fuzzers find both vulnerabilities, buffer overflow and string format, whilst Sulley only finds the buffer overflow. Sulley also finds less vulnerabilities in general, this is not necessarily bad because the main goal is to find all unique vulnerabilities, but it shows that Sulley is less able to search the input space. In general strategy 1 is the best strategy for finding vulnerabilities because it found all unique vulnerabilities and it also found the most vulnerabilities. Table 4.2 also shows that strategy 1 in normal mode performs better than strategy 1 in random or no feedback mode. This indicates a possible correlation between code coverage and finding vulnerabilities since the latter two modes do not feed the genetic algorithm proper coverage information. The same holds for strategy 3 and the random and no feedback mode.

The figure 4.2a shows how the strategies compare to each other on basic block coverage. For basic block coverage the picture is unclear on the best strategy. Strategy 3 outgrows the other fuzzing strategies but stops early. Strategy 4 is for a long time equal to strategy 2 but grows fast at the end, reaching the same number of basic blocks as strategy 3. Like strategy 4, strategy 2 has a fast growth in the number of basic blocks at the end. Strategy 1 is a bit slow in the beginning but at the end has the most basic blocks covered. Sulley performs worse than other strategies. This is possibly due to the fact that Sulley does not try to optimize basic block coverage.

Figure 4.2b shows how the strategies compare on edge coverage. Here the picture is more clear. Strategy 1 outperforms the other strategies in terms of edge coverage. Strategy 3 and 4 perform less than strategy 1 but better than strategy 2. Strategy 2 performs less than other strategies but better than Sulley. Sulley does again not perform well which is likely due to the fact that Sulley does not try to optimize edge coverage.

Figure 4.3a shows how strategy 1 in different modes compare to each other for basic block coverage. Expected is that strategy 1 in normal mode performs better than strategy 1 in random or no feedback mode. The results show that the normal mode is only better after about 5000 test-cases. This is likely due to the fact that the focus is slightly favored on edges instead of basic blocks. Figure 4.3b shows how strategy 1 in different modes compare to each other in terms of edge coverage. The figure is clear. Strategy 1 is better than the random and no feedback modes. The results show that strategy 1 in normal mode is successful in finding new edges whilst the other modes are not. The differences between the random and no feedback mode are minimal.

In figures 4.4a and 4.4b basic block coverage and edge coverage are plotted against the number of test-cases for strategy 3 in different modes. It shows the same pattern as strategy 1 for different modes, although the differences between them are smaller. This makes sense because all three modes were also roughly equal in finding vulnerabilities. There are several reasons why this could happen. First of all it could be because of the randomness involved in fuzzing. Another reason could be that the strategy is flawed or implemented wrong.

The results for Vuln_server 1 for the second execution are about the same. In both executions the fuzzers found about the same number of vulnerabilities and all found the two unique vulnerabilities. All fuzzers have about the same coverage in both runs except for strategy 1. For strategy 1 in normal mode the results where only improving over the random and no feedback mode after about 5000 test-cases 4.3a. In the second execution strategy 1 in normal mode immediately improves over the other modes. Another difference is that in the second execution the no feedback mode is better than the random mode whilst in the first execution the random mode is better than the no feedback mode. The small differences between executions can be attributed to the random elements of MyFuzzer.

From the results for Vuln_server 1 in experiment 1 the following preliminary conclusions can be taken:

- Strategy 1 finds the most vulnerabilities.

- All instances of MyFuzzer found the unique vulnerabilities.
- Sulley did not find the string format vulnerability.
- Strategy 1 has in the end the best basic block coverage.
- Strategy 1 has the best edge coverage.
- Sulley has poor coverage most likely because it does not have the incentive to get a good coverage.
- The genetic algorithm helps with creating better test-cases because fuzzers in alternative modes perform less than the normal mode. The normal mode has better code coverage and finds vulnerabilities more often.
- The second execution validates the results from the first execution.

4.4.1.2 Vuln_server 2

Vuln_server 2 was also executed twice for experiment 1. The results for the first execution can be seen in 4.3.1.2. The results for the second execution can be provided upon request.

As can seen in table 4.3 all strategies find the three unique vulnerabilities except strategy 4 and Sulley. Sulley does not find the string format vulnerability in the ”REVERSE” command whilst strategy 4 does not find the vulnerability in the ”COPY” command. It can also be seen that both strategy 4 and Sulley do not find a lot of vulnerabilities. Strategy 3 performed better and found all vulnerabilities. Strategy 1 and 2 where the best and found a vulnerability once every 50 test-cases.

As can seen in table 4.3 strategy 1 in normal mode finds more vulnerabilities over time than strategy 1 in no feedback or random mode, although all three modes found all unique vulnerabilities. This shows the correlation between code coverage and finding vulnerabilities since the last two modes do not give the genetic algorithm proper feedback. Another possible explanation could be that the normal mode has a bias towards test-cases which find vulnerabilities by using them again. That is however unfeasible because those test-cases get the lowest possible score and are thus not likely to be selected to be reused. No difference between strategy 4 in normal mode and strategy 4 in no feedback or random mode could be observed. For strategy 2 the same pattern as for strategy 1 could be observed although the results differ less.

Figure 4.5a clearly shows that strategy 1 is the best strategy in finding new basic blocks. Strategy 2 performs less well but still reaches the same number of basic blocks as strategy 1 eventually. Strategy 4 and Sulley are roughly equal but stop earlier than other fuzzers. Strategy 3 initially does not perform well but because it executes more test-cases than other fuzzers it eventually reaches the same number of basic blocks than strategy 1 and 2. Figure 4.5b shows again that strategy 1 has the best results in terms of edge coverage. This time strategy 2 is a clear second. The other three strategies perform less, but because of the number of test-cases strategy 3 reaches about the same number as edges as strategy 1. Sulley performs the worst.

Figure 4.6a shows the basic block coverage results for strategy 1 in multiple modes. The figure shows that the genetic algorithm works because if MyFuzzer gets wrong or no feedback it doesn't know what to look for. Figure 4.6b shows the results for edge coverage for strategy 1 in multiple modes. Again it shows that the genetic algorithm does its work because the normal mode outperforms the no feedback and random mode.

For strategy 2 the coverage results can be seen in figure 4.7a and 4.7b. Noteworthy is that strategy 2 in no feedback mode has a better edge coverage than strategy 2 in normal mode whilst for basic block coverage it is the other way around.

For strategy 4 the coverage results can be seen in figure 4.8a and 4.8b. Strategy 4 and modes follow the same patterns as strategy 1.

There are more differences between the two executions for Vuln_server 2 than for Vuln_server 1. First of all the fuzzers in the first execution find more vulnerabilities than the fuzzers in the second execution. The only difference which could have benefited the first execution is that there is more delay between test-cases. As for coverage the results for both executions where about the same. Only the coverage results for strategy 1 where somewhat strange. In the first execution the graphs for code coverage against the number of test-cases for strategy 1 in different modes 4.6a 4.6b showed clearly that the normal mode was better than the other modes. In the second execution the basic block coverage for the normal mode is worse by about 100 basic blocks than the other modes whilst the edge coverage for the normal mode is better. There are two possible reasons for this. First of all it could be due to the random nature of MyFuzzer. Secondly it could be because of the fact that for this experiment the focus was slightly more on edge coverage.

From the results for Vuln_server 2 the following conclusions can be taken:

- Sulley and strategy 4 did not find all vulnerabilities and had poor search results.
- Strategy 1 and 2 found the most vulnerabilities.
- The no feedback and random mode for strategy 1 performed less in the first execution than in the second execution.
- Little or no difference between strategy 2 in normal mode, random mode or no

feedback in terms of vulnerabilities.

- Strategy 1 has the best basic block coverage.
- The genetic algorithm works for strategy 1 because it has a better code coverage than strategy 1 in random or no feedback mode.
- Better coverage does not always correlate with more vulnerabilities.

4.4.1.2.1 Conclusion Experiment 1 showed that strategy 1 generally gives the best results. That is why it will be used in experiment 2 and 3. The results for both executions where about the same for both targets. The differences between the same strategy in different modes shows that the code coverage information helps the genetic algorithm to select the best test-cases. This is especially true for fuzzers using strategy 1. The results also show that MyFuzzer has an edge over Sulley because Sulley is not able to find the string format vulnerability and also has less of the code covered.

4.4.2 Experiment 2

Experiment 2 has been executed thrice on the Proftpd target and once on both Vuln_server applications and EasyFTP.

4.4.2.1 EasyFTP

For EasyFTP two executables where monitored. The first executable is the actual FTP server and the second is the graphical user interface. The results for the interface are of minor importance because the fuzzer only communicates with the FTP server. Therefore the results for the interface executable are not shown in the results section.

Table 4.4 shows how many vulnerabilities each fuzzer found in EasyFTP. It can be seen that strategy 1 in no feedback mode and Sulley find the most vulnerabilities. Noteworthy is that strategy 1 in normal mode only found six unique vulnerabilities whilst the other two fuzzers found seven. There are several possible reasons for this. First of all this could be because of the generation phase of MyFuzzer. For each node an initial set of test-cases has to be generated. In the target communication specification for EasyFTP there is one node which can take about 40 command values. It could be that the generator did not generate enough test-cases for one of the commands. A second reason why this happened could be simply because of the random elements of MyFuzzer.

In figure 4.9a the results for the server executable can be seen. As can be seen Strategy 1 outperforms the other two by a slight margin. Sulley had an increase in basic block coverage, however at its peak the stop criterion was reached. The stop criterion for Sulley is that all test-cases have been sent. It remains unknown if Sulley would continue to increase if it was given 20000 more test-cases. The jumps in the figures are due to the fact that a new node is fuzzed and so new basic blocks are reached.

The edge coverage for the server executable can be seen in 4.9b. As expected strategy 1 is the best. Sulley performs significantly worse than the other fuzzers. This is probably due to the fact that Sulley is deterministic. The lack of a genetic algorithm optimizing for code coverage cannot be the sole reason of Sulley performing less because strategy 1 in no feedback mode does not receive proper feedback and still manages to get a higher edge coverage than Sulley. This is evidence that random fuzzing leads to better edge coverage in comparison to deterministic fuzzing.

The results for EasyFTP in experiment 2 show some interesting things. First of all there is no correlation between coverage and finding vulnerabilities because strategy 1 has the best coverage whilst it does not find all seven unique vulnerabilities. The explanation for this might be that the differences between the fuzzers are not significant and that code coverage does not give guarantees about finding vulnerabilities. Another interesting result is that Sulley does have a good basic block coverage but is significantly worse for edge coverage. This can be attributed to the lack of a genetic algorithm for Sulley. The fact that Sulley is able to find basic blocks and not a lot edges can be attributed to the fact that finding edges is more difficult to happen by chance. The third interesting result is that strategy 1 in no feedback mode also performs well.

4.4.2.2 Proftpd

Proftpd was executed thrice for experiment 2. The results for the first execution can be seen in 4.3.2.2. There where some difficulties with getting Sulley working for Proftpd. The main problem was that Sulley became incredibly slow, sometimes taking more than 20 seconds per test-case. The second problem was that Sulley had a lot of sudden crashes which required a manual reset. Sulley was included in the third execution.

For Proftpd no vulnerabilities where found in all three executions. According to exploit-db [30] there is an vulnerability present in Proftpd 1.3.3a. For some reason none of the fuzzers was able to find this vulnerability.

The code coverage figures for execution 1 4.10a and4.10b show some strange jumps. Execution 2 and 3 examined if these jumps where because of ASLR or libraries associated with Proftpd respectively. But the jumps where also observed in execution 2 and 3. Because the jumps are very large, even larger than the basic blocks executed in the initialization of the target, it is doubtful if the jump is caused by a test-case. It could also be that there is another cause for the jump like operating system intervention or occasionally executed code. If the latter is the case the jump would be an anomaly. Jumps which happen at the same time for all fuzzers are likely due to the fact that they reached a new node and therefore execute a significant amount of new basic blocks. Jumps which only happen in one fuzzer look more suspicious. It is assumed that the jumps happening in all fuzzers are caused by a test-case whilst jumps which only happen in one fuzzer are anomalous. This is just an assumption and not a fact. More research is needed to establish the cause of these jumps. The jumps mights also be signs that Proftpd is not a suitable target for MyFuzzer.

The results for Proftpd in experiment 2 lead to the following conclusions.

- None of the fuzzers found a vulnerability in Proftpd.
- The coverage graphs contain several jumps. More research is needed to establish what is causing them.
- The results indicate the accuracy of MyFuzzer might be target dependent.

4.4.2.3 Vuln_server 1

The results for Vuln_server 1 can be found in section 4.3.2.3. Experiment 3 was executed once for Vuln_server 1.

In table 4.5 it can be seen that Sulley does not find a lot of vulnerabilities. Sulley also does not find the string format vulnerability. The results show that the normal mode is slightly better in finding vulnerabilities.

The basic block graph, displayed in figure 4.11a, shows that all fuzzers except Sulley finish on the same number of basic blocks. Strategy 1 is however the first to reach this number. Random feedback looks to be better then no feedback. What is also seen is that every once in a while new basic blocks are found but most of the time no new basic block is found. This is likely due to the fact that it is a very small program and that some block are only reached through rare inputs.

The edge coverage graph, figure 4.11b, shows the same as the basic blocks. Only Sulley performs significantly worse than the other strategies. Although the differences with basic blocks were minimal, the results for edge coverage are more significant. Strategy 1 has a clear advantage in edge coverage over other fuzzers.

The results for Vuln_server 1 in experiment 2 can be used to conclude the following:

- Strategy 1 in normal mode finds the most vulnerabilities.
- Sulley does not find all vulnerabilities.
- Strategy 1 reaches maximum number of basic blocks first.
- Sulley has a worse basic block and edge coverage.
- Results show correlation between coverage and vulnerabilities.
- Results show that the genetic algorithm works.

4.4.2.4 Vuln_server 2

The results for Vuln_server can be seen in section 4.3.2.4.

In table 4.6 the vulnerability results can be found. Like before Sulley does not find the string format vulnerability. The other methods find all the unique vulnerabilities. Strategy 1 in normal mode has an advantage over other methods because it is able to find vulnerabilities more regularly.

In figure 4.12a it can be seen that strategy 1 has an edge over other fuzzers in terms of basic block coverage. The difference between strategy 1 in normal mode and strategy 1 in random mode or no feedback mode is minimal. Sulley on the other hand performs significantly worse. It could be that, like Vuln_server 1, the executable itself is to small to solely rely upon since it only consists of about 130 basic blocks.

For edge coverage, which can be seen in figure 4.12b, the same trend as with basic blocks is observed. Only this time the differences are bigger. A possible explanation for this is that the fuzzers focus slightly more on edge coverage, which might explain why strategy 1 in normal mode has a higher edge coverage.

The result for Vuln_server 2 in experiment 2 can be used to make the following conclusions:

- Strategy 1 finds the most vulnerabilities.
- Sulley is not able to find all vulnerabilities.
- All strategies arrive at a basic block count of ca 132, which means that Vuln_server 2 likely consists of 132 basic blocks.
- Strategy 1 in normal mode is a bit earlier than other methods in finding new basic blocks.
- Strategy 1 in normal mode has a better performance on edge coverage.
- Results show a correlation between coverage and vulnerabilities.
- Results show the effectiveness of the genetic algorithm.

4.4.2.5 Conclusion

The results for Vuln_server 1 and 2 validate the conclusions from experiment 1. The results for EasyFTP showed that strategy 1 in normal mode has the best code coverage but did not find all vulnerabilities. It shows that the genetic algorithm does its work but that this does not necessarily lead to more vulnerabilities being found. For Proftpd 1.3.3a the results were different. None of the fuzzers found the vulnerability in Proftpd. The code coverage results also showed several large increases in basic block coverage. Two potential causes where explored in execution 2 and 3, but they did not solve the problem. The most likely causes of these increases, now that ASLR and target dependencies have been ruled out, is that either a new node is being found, in which the increase for all fuzzers should be roughly equal, or it is an anomaly. Regardless of what causes the sudden increases the results for Proftpd should be used carefully.

4.4.3 Experiment 3

In experiment 3 four different instances of MyFuzzer strategy 1 where compared with each other. Every know and than results from previous experiments are used as a supplement, simply because they are available and compatible.

4.4.3.1 Vuln_server 1

In experiment 3 Vuln_server 1 was executed thrice. The results for the first execution can be seen in section 4.3.3.1.

A focus on basic block coverage delivers the best results as can be seen in table 4.7. It finds the most vulnerabilities, however it has to be stated that all fuzzers find all unique vulnerabilities.

The coverage data can be seen in figure 4.13a and 4.13b. What is immediate noticed is that pure methods, where the focus is either on basic blocks or edges, perform the best. The default settings of strategy 1 (slight edge towards edge coverage) also does not do bad, but it might be that there where slight differences between executions because the results for strategy 1 where obtained in an earlier experiment. Making the genetic algorithm dependent on multiple coverage metrics seems to decrease performance. A reason why the focus on basic blocks gives the best results is that finding a new basic block also implies a new edge. Using multiple metrics might confuse the genetic algorithm since it has multiple goals which could be in conflict with each other.

The results for the second execution showed no significant differences in both vulnerabilities and code coverage. This shows that ASLR has a minimal influence on the results. The third execution showed different results for both basic block coverage. A full focus on either edge or basic block coverage leads to less code coverage whilst fuzzers depending on both edge and basic block coverage metrics perform better. This might me evidence that the coverage information code coverage of dependencies must also be taken into account. Another reason for this reversal can be that the executable of Vuln_server 1 is to small to create conclusions since it only consists of 86 basic blocks.

The results for Vuln_server 1 in experiment 3 can be used to conclude the following:

- Full focus on basic block coverage delivers the best basic block and edge coverage. Probably because a new basic block automatically means a new edge as well.
- Full focus on basic block coverage found the most vulnerabilities. This is evidence for the correlation between coverage and finding vulnerabilities.
- Making the fitness function dependent on both basic block and edge coverage metrics usually makes the fuzzer perform less both in terms of finding vulnerabilities and coverage.
- When only the coverage of the target executables are taken into account the results are different.

4.4.3.2 Vuln_server 2

Vuln_server 2 was also executed thrice. The results for the first execution of Vuln_server 2 can be seen in section 4.3.3.2.

Table 4.8 shows how many vulnerabilities have been found. As can be seen all fuzzers find the unique vulnerabilities. The fuzzer which focuses on basic blocks finds the most vulnerabilities. In contrast to Vuln_server 1, a focus on edges does not perform well.

The figures 4.14a and 4.14b display basic block and edge coverage respectively. The figure for basic block coverage shows that initially the default strategy 1 performs well. After a while however the fuzzer which focuses on basic blocks takes over and becomes the fuzzer with the best basic block coverage. The other fuzzers end relatively close together. Edge coverage shows again that a focus on basic blocks delivers the best edge coverage. Again this is possibly due to the fact that a new basic block implies at least 1 new edge. A more strange result is that the fuzzer focusing on edge coverage does not perform well in this case.

There are big differences between runs 1, 2 and 3 for Vuln_server 2. In run 1 the fuzzer with a fitness function focusing on basic block coverage gets a higher overall basic block coverage. Because it has found more basic blocks its edge coverage is also higher. The results for run 1 correspond with the results from Vuln_server 1, which indicate that a focus on one coverage metric (basic blocks) works better. In run 2 the focus on basic bock performs significantly less well than in run 1. It is unclear why this happens. In run 3 the results for all fuzzers are close together. A focus on basic block coverage seems to have a slight edge over other fuzzers, confirming earlier observations, but the margins are too narrow. In terms of vulnerabilities, each fuzzer found all unique vulnerabilities.

The results for Vuln_server 2 can be used to make the following preliminary conclusions:

- Strategy 1 with a focus on finding new basic blocks finds the most vulnerabilities.
- Strategy 1 does also perform well coverage wise.
- Full focus on basic blocks delivers the best coverage.
- Focus on edge coverage does not give as good results as in Vuln_server 1.

4.4.3.3 Proftpd

Proftpd was executed twice for experiment 3. The results for Proftpd can be seen in section 4.3.3.3.

Like in experiment 2, none of the fuzzers found a vulnerability in protfptd 1.3.3a.

In figure 4.15a and figure 4.15b the coverage results for Proftpd can be observed. The basic block coverage graph contains several big increments at certain places. The results become difficult to analyze because of these increments. If the increments in basic blocks are caused by the test-cases send by the fuzzer than strategy 1 with a full focus on basic block coverage delivers the best coverage results, both for basic block coverage and edge coverage. If the increments in basic block coverage are not caused by the test-cases than it remains unclear which fuzzer had the best performance.

In the second execution the results were about the same. For execution second the results contained again some sudden increments in basic block coverage. Strategy 1 with a focus on basic blocks in the end had a slight advantage in terms of basic block and edge coverage over the other fuzzers.

The big question is what causes the jumps in basic block coverage. Nonetheless some conclusions can be taken from the results for Proftpd in experiment 3:

- No vulnerabilities found.

- Basic block coverage data contains sudden increments which makes the analyzes of the result more difficult.
- Results for this target should be used carefully.

4.4.3.4 EasyFTP

The results for EasyFTP can be seen in section 4.3.3.4.

Strategy 1 focusing on basic block coverage finds the most vulnerabilities as can be seen in table 4.9. The fuzzer with a focus on edge coverage is able to find all seven unique vulnerabilities. Half edge and half basic block only finds 5 unique vulnerabilities and 7 vulnerabilities in total.

EasyFTP consists of two executables and for both the code coverage is kept track of. However the server executable is the most important one because that is the executable which handles all FTP requests. The console executable handles is for the GUI and is not directly fuzzed. The code coverage graphs for the server executable can be seen in figures 4.16a and 4.16b. The first thing noticed is that there is a very big jump in basic block coverage at around 8000 test-cases. This jump is present in every fuzzer and the amount is about the same for every fuzzer so most likely a new important node is being fuzzed. The results are too close together to make solid conclusions. The focus on basic block coverage seems to find basic blocks more easily, but in the end all fuzzers arrive at about the same number of basic blocks. The same is true for the edge coverage. The fuzzer with a focus on edge coverage seems to have a slight edge over other methods.

The result for EasyFTP can be used to construct the following preliminary conclusions:

- Only one of three methods was able to find all seven vulnerabilities.
- A focus on edges gives a slight edge in edge coverage over other methods.
- A focus on basic blocks gives a slight edge in basic block coverage over other methods.
- Results show no clear correlation between different code coverage metrics and vulnerabilities.
- Full focus on one metric seems better than using multiple metrics.

4.4.3.5 Conclusion

In most instances the data shows that a focus on basic block coverage gets the best results. There are exceptions to this. The second execution of Vuln_server 2 has a completely different results and does not think a focus on basic block coverage is the best. It could be that the experiment was executed wrongly since both Vuln_server applications had similar results for different executions in experiment 1 and 2. Another possible reason might be that it was an anomaly. Also Vuln_server 1 had similar results for different executions in experiment 3. On the other side Vuln_server 1, Proftpd and EasyFTP indicate that a focus on basic block coverage delivers the best results.

Chapter 5

Discussion

There are several things which have influence on the results. First of all the mutation engine uses random mutations. This means that two identical fuzzers might have different results for a given target because the first fuzzer might get more beneficial mutations. This is usually solved by repeating the experiment multiple times. The second influence is the target itself. The target might sometimes be non-deterministic. One example is a target which uses a random command to generate integers. This means that a non-deterministic target might execute different pieces of code for the same test-case. The operating system might also have an influence on the results because the target is usually depending on the operating system. To mitigate this the experiments where executed multiple times. Because of time constraints the experiments could only be repeated three times.

The combination of fuzzing with genetic algorithms and code coverage is relevant because in the academic literature there are quite a few suggestions that the combination might fit nicely together. In [21] a survey about state of the art fuzzing is done and they conclude that future fuzzers will likely incorporate gray-box methods, like genetic algorithms, to improve accuracy. In [5] code coverage and search algorithms are an integral part of the proposed solution. The use of genetic algorithms in combination with fuzzing is also given in the following research papers [10, 14, 17]. So there are quite a few resources which highlight the usefulness of genetic algorithms. Some have also put it to practice [10, 17]. Optimizing the genetic algorithm for maximizing code coverage is also proposed in the literature [5].

There are two articles which show that a fuzzer using a genetic algorithm performs better than a fuzzer without a genetic algorithm. In [18] the GAFuzzing tool, which uses genetic algorithms, is compared with a random fuzzer on a target called VulPro. The results show that GAFuzzing performs better than random fuzzing. In [17] several fuzzers with genetic algorithms are compared with Peach and Mangle. Peach and Mangle both do not use genetic algorithms. The target in [17] was the Mac OS X application preview. The results showed that a fuzzer using a genetic algorithm generally had better results than Mangle or Peach. The hypothesis that a genetic algorithm improves fuzzing is confirmed by the results from this project. Sulley does not find the string format vulnerability in the vuln_server applications and has generally lower code coverage than MyFuzzer.

To my knowledge there is only one paper [17] where different metrics for the fitness function are evaluated. Some other sources which can be used for comparison are how well known fuzzers like AFL, EFS and GAFuzzing instruct the fitness function. To the author's knowledge the choices for fitness functions in these fuzzers are not based on research. AFL uses edges between basic blocks as the main metric for the fitness function [16]. GAFuzzing [18] inserts predicates into the target and uses predicate coverage as a metric. Test-case fitness for EFS [10] is simply dependent on the impact of the test-case on the target, e.g. how many basic blocks did the test-case trigger. The results from this project show that a focus on finding new basic blocks generally delivers the best results. This is in contrast with AFL which uses edges as the main code coverage metric. Comparing the results for MyFuzzer with EFS is difficult since MyFuzzer is never fully reliant on the impact of test-cases. Only in experiment 3 there is a MyFuzzer instance which uses impact as one of the three metrics and no significant improvements for that instance over others could be observed. It is difficult to compare the result of [17] with the results for this project since [17] does not use code coverage as a fitness function metric.

MyFuzzer was executed on two FTP server applications and two dummy applications. The impact of MyFuzzer on other types of applications is thus unknown. But based on the results presented in this report it can be assumed that MyFuzzer also works for other server applications like SSH. It also indicates that MyFuzzer should be able to work on the software systems developed at Thales Hengelo. Further experimentation is needed to validate if MyFuzzer delivers the same results for other targets.

Chapter 6

Future Work

Whilst the solution presented in this thesis had significant results there are a lot of aspects which can be improved. The future work can be split up into two categories: practical and research. The section about practical future work gives some pointers about how MyFuzzer can be improved to be more useful in practical settings. The research section gives some possible extensions of the research presented in this report.

6.1 Practical

In addition to code refactoring and adding more documentation some practical actions can be taken to improve MyFuzzer.

6.1.1 GUI

MyFuzzer has a lot of parameters like mutation rates, fuzzing strategy and metric weights. Right now these parameters, together with the target communication specification, are given in a python file. So if someone wants to create a setup for a new target a new python file has to be written. A solution to this is to use a GUI where everything can be configured. This has two main benefits. First of all no programming skills are required to fuzz a new target. Employees without any programming skills can now configure MyFuzzer for the target they want. Secondly a GUI also helps in preventing setup mistakes.

6.1.2 Central Point of Configuration

Right now several steps have to be done to start MyFuzzer. First of all the monitors on the target system need to be executed. These monitors also take some parameters as input. After the monitors are running the fuzzer has to be started. So the initialization of MyFuzzer consists of several steps on different systems. These steps are not necessarily difficult but they are error prone. A better solution is to move all settings to a central point. A good central point would be the GUI discussed in 6.1.1. Note that the GUI must be able to access the target system so some kind of interaction is needed between target system and the fuzzing system to setup a connection.

6.1.3 Attack Libraries

Right now the attack libraries are only given on primitive level. The attack libraries can be extended on two points. First of all new attack vectors for certain primitives could be added to make it more complete. Another possibility is to introduce attack libraries on higher levels. For example an attack library on block level could contain potentially dangerous blocks.

6.2 Research

6.2.1 Mutation Rates

Research about optimal mutation rates is a reoccurring theme in genetic algorithms. If the mutation rates are too high good traits of test-cases will not be preserved and if the mutation rates are too low the test-cases will look too much like each other. More research about mutation rates could give indications about which mutation rates are optimal and if mutation rates are target or primitive dependable.

6.2.2 Code Coverage using Source-code

Right now MyFuzzer works on runtime code coverage metrics, so no source-code is needed. This project could also been done using code coverage metrics using source-code. For example gcov could be used to obtain coverage information. The fuzzing process stays the same. The expectations are that using source-code the accuracy would increase because more information about the target is known. It is also likely that the fuzzing process becomes faster because no additional virtualization layer, like DynamoRIO, is needed. MyFuzzer could be extended with functionality for fuzzing using code coverage metrics on source-code level. A comparison between runtime code coverage and code coverage on the source-code could help determining which method is better.

6.2.3 Other Metrics

For this project the main metrics used by the fitness function are code coverage metrics. There are however more metrics which could be used like number of variables, number of functions and number of vulnerability patterns. These metrics might be more useful to the genetic algorithm than code coverage metrics. Experimenting with different metrics might show that some metrics are more important for the genetic algorithm than others.

6.2.4 Multiple Populations

In [10] there are multiple parallel populations. This might be a good idea for the following reason. Like with natural selection in biology, populations are driven towards a solution. In the long run this solution might not be optimal. Multiple populations would probably show different behavior, so if a population does not perform well any more it can be removed. Research could be done comparing fuzzers with one population against fuzzers which use multiple populations.

6.2.5 Multiple Targets and Different Configurations

The number of configurations for MyFuzzer is large and the number of potential targets is also large. To validate the results for this project it is a good idea to run MyFuzzer in the same or different configuration on new targets.

Chapter 7

Conclusion

The results and analysis presented in the experiments and discussion chapters can be used to answer the research questions given in the introduction 1. Each subquestion will be answered here:

How does MyFuzzer compare to existing fuzzers in terms of finding vulnerabilities and code coverage:
As can be seen in the results of experiment 1 and 2, MyFuzzer instances perform usually better than Sulley in terms of coverage. In terms of vulnerabilities no significant difference can be seen between Sulley and MyFuzzer except for the fact that MyFuzzer finds vulnerabilities quicker. In Vuln_server 1 and 2 Sulley does not find the string format vulnerability. On the other hand MyFuzzer does not always find the same vulnerabilities for EasyFTP than Sulley.

How does the choice of metrics for the fitness function influence the accuracy of MyFuzzer:
In experiment 3 four fuzzers with different fitness functions were tested. The first fitness function focused on finding new basic blocks, the second on finding new edges, the third gave both new edges and new basic blocks equal importance and the last one gave the impact of a test-case, new basic blocks and new edges equal importance. From experiment 3 it was learned that most of the time a full focus on new basic blocks delivers the best results for basic block and edge coverage. Combinations of multiple metrics generally performed less. In terms of finding vulnerabilities no significant difference could be observed. In general the fuzzers for experiment 3 found the same set of vulnerabilities.

How does the choice of fuzzing strategy influence MyFuzzer in terms of finding vulnerabilities and code coverage: The last subquestion is not the most important and cannot be answered with the most confidence. In general strategy 1 seems to be the best strategy for MyFuzzer, which was enough incentive to use strategy 1 as the main strategy.

In conclusion, this report has shown two things. First of all it has shown that it is possible to create a fuzzer driven by a genetic algorithm without requiring access to the source-code. Secondly the research part of this project confirms that a fuzzer with a genetic algorithm improves over a fuzzer without a genetic algorithm. The research also indicates that a genetic algorithm which tries to maximize the number of basic blocks tested gives the best results.

Additional experiments with different types of targets and different parameter settings can be done to validate the results obtained in this project.

Bibliography

[1] Fuzzing. `https://www.owasp.org/index.php/Fuzzing`, 2016. [Online; accessed 04-January-2017].

[2] Dynamorio. `http://dynamorio.org`, 2017. [Online; accessed 19-April-2017].

[3] Peach fuzzer. `http://www.peachfuzzer.com/`, 2017. [Online; accessed 25-January-2017].

[4] B. H. Arabi. Solving np-complete problems using genetic algorithms. In *2016 UKSim-AMSS 18th International Conference on Computer Modelling and Simulation (UKSim)*, pages 43–48, April 2016.

[5] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 427–430, March 2011.

[6] cacalabs. zzuf. `http://caca.zoy.org/wiki/zzuf`, 2016. [Online; accessed 25-January-2017].

[7] J. Cai, S. Yang, J. Men, and J. He. Automatic software vulnerability detection based on guided deep fuzzing. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pages 231–234, June 2014.

[8] J. Cai, P. Zou, D. Xiong, and J. He. A guided fuzzing approach for security testing of network protocol software. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 726–729, Sept 2015.

[9] Steve Cornett. Code coverage analysis. `http://www.bullseye.com/coverage.html`, 2014. [Online; accessed 25-January-2017].

[10] J. Demott, R.J. Enbody, and W. Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. Oct 2007.

[11] Parul Garg. Fuzzing - mutation vs. generation. http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation, 2012. [Online; accessed 03-January-2017].

[12] Dan Geer. The physics of digital law. In *CyberCrime and Digital Law Enforcement Conference*, March 2004.

[13] Intel. Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2017. [Online; accessed 19-April-2017].

[14] E. Jskel. Genetic algorithm in code coverage guided fuzz testing. Master's thesis, Dec 2015.

[15] Richard Kissel. Glossary of key information security terms. Technical report, National Institute of Standards and Technology, 2013.

[16] lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl, 2016. [Online; accessed 05-January-2017].

[17] Roger Lee, Accepted For The Council, Carolyn R. Hodges, and Roger Lee Seagle. A framework for file format fuzzing with genetic algorithms. 2012.

[18] G. H. Liu, G. Wu, Z. Tao, J. M. Shuai, and Z. C. Tang. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *2008 Third International Conference on Convergence and Hybrid Information Technology*, volume 2, pages 491–497, Nov 2008.

[19] Mauro Pezz Luciano Baresi. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 2006.

[20] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional., first edition, 2006.

[21] Richard Mcnally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: The state of the art. 2017.

[22] Ruth Breu Matthias Buchler Alexander Pretschner Michael Felderer, Philipp Zech. Model-based security testing: a taxonomy and systematic classification. *Software Testing, Verification and Reliability*, 2016.

[23] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, first edition, 1999.

[24] NIST. National vulnerability database. https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cvss_version=3. [Online; accessed 25-January-2017.

[25] Jiantao Pan. Software testing. https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/, 1999. [Online; accessed 25-January-2017].

[26] Jeff Offutt Paul Amman. *Introduction to Software Testing*. Cambridge University Press, second edition, 2008.

[27] Ryan Sears Pedram Amini, Aaron Portnoy. Fuzzing - application and file fuzzing. https://github.com/OpenRCE/sulley, 2016. [Online; accessed 03-January-2017].

[28] Tsong Yueh Chen John Clark Myra B. Cohen Wolfgang Grieskamp Mark Harman Mary Jean Harrold Phil McMinn Antonia Bertolino J. Jenny Li Hong Zhu Saswat Anand, Edmund K. Burke. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013.

[29] Bruce Schneier. Complexity the worst enemy of security. https://www.schneier.com/news/archives/2012/12/complexity_the_worst.html, 2012. [Online; accessed 19-April-2017].

[30] Offensive Security. Offensive securitys exploit database archive. https://www.exploit-db.com/. [Online; accessed 14-October-2017.

[31] B. Shuai, M. Li, H. Li, Q. Zhang, and C. Tang. Software vulnerability detection using genetic algorithm and dynamic taint analysis. In *2013 3rd International Conference on Consumer Electronics, Communications and Networks*, pages 589–593, Nov 2013.

[32] Huang Song, Wang Liang, Zheng Changyou, and H. Yu. A software security testing method based on typical defects. In *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, volume 5, pages V5–150–V5–153, Oct 2010.

[33] Wikipedia. Genetic algorithm. https://en.wikipedia.org/wiki/Genetic_algorithm. [Online; accessed 18-April-2017].