Using Grammar Extracted from Sample Input to Generate Effective Fuzzing Files

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies at

University of Idaho

by

Hamad Al Salem

Approved by:

Major Professor: **Jia Song**, Ph.D.

Committee Members: James Alves-Foss, Ph.D.; Terence Soule, Ph.D.; Xiaogang Ma, Ph.D.

Department Administrator: Terence Soule, Ph.D.

December 2021

Abstract

Software testing is an important step in the software development life cycle. It focuses on testing software functionalities, finding vulnerabilities, and assuring the software is executing as expected. Fuzzing is a software testing technique which feeds random input to programs and monitors for abnormal behaviors such as a program crash. Fuzzing can be automated and does not require access to the source code compared to manually reviewing the source code which requires a huge amount of time and cost. It can trigger vulnerabilities that the programmers overlooked while programming, such as buffer overflow, off by one error, etc. One of the limitations of fuzzing is that most programs require highly structured input or certain input patterns and therefore the fuzz testing may be terminated at an early stage of program execution because of not meeting the input format requirements. Some previous studies resolve this problem by manually creating program specific input grammars to help guide fuzzing, which is tedious, error prone, and time consuming. However, this solution cannot work efficiently when testing multiple programs which require different input patterns. To solve this problem, a general grammar-based fuzzing technique is proposed and developed in this dissertation. The new fuzzer can extract grammar from the sample input files of a program, and then generate effective fuzzing files based on the grammar. This fuzzing tool is able to work with different programs by extracting grammars from them automatically and hence generating program specific fuzzing files. The goal of this research includes developing an algorithm to extract grammars from sample input files, generating effective fuzzing files to test the programs, and implementing a fuzzing tool using Python programming language. The main contribution of this research is helping software developers and security experts in revealing vulnerabilities in various programs automatically by using the developed

fuzzing tool.

Acknowledgments

First and Foremost, I must thank Allah the almighty for giving me the ability and success to accomplish this dissertation. Without Allah's blessings and favour, this dissertation would not be achieved.

I would like to thank my advisor, Dr. Jia Song, for her support, encouragement, and patient guidance throughout my graduate studies. Her supervision and support drove me to conduct the research, gaining a wide knowledge, writing, and completing this dissertation. Without her guidance through out the days and years, I would not imagine finishing or conducting the research. I am proud of being one of Dr. Song's students.

I would also like to thank Dr. James Alves-Foss, who gave me a hand and valuable advises during my studies in UI. Also, I would like to thank him for his valuable comments on my dissertation

I would also like to thank my other committee members, Dr. Soule, and Dr. Ma for their support and valuable comments on my dissertation.

I would like to thank all my instructors for their hard work and dedication in providing me with a comprehensive and valuable education.

I would like to thank the staff in department of Computer Science for their help during my study in the department.

I would like to thank all the staff of the College of Graduate Studies for their help and support throughout my study at UI.

I would like to thank many other friends who made my life at UI very enjoyable.

Finally, I would like to thank Saudi Culture Mission and Najran University for their support and funding my studies in UI and finishing this dissertation.

Dedication

I dedicated this work to my dearest parents, my father ***ALI ABOSAQ*** *and my mother* ***QAINAH ABOSAQ****. I do not have the words to express my gratitude and appreciation for their enormous sacrifice, patience, support, encouragement, care, interesting, trust, and raising me.*

Also, This work is dedicated to my dearest grandmother ***Refah Abosaq****. I also dedicated this work to my dearest love my wife for her support, patience, and taking care of our children during my studies. Also, I dedicated this work to my children, my brothers, and my sisters for their understanding, support, encouragement, and love which have helped me make this dissertation a reality.*

Contents

List of Tables

List of Figures

Chapter 1: Introduction

Nowadays, software security plays an important role in the field of computer science. There are millions and millions of software products on the market which provide services to end users. The security of the software has become more and more critical because people rely on some software to accomplish certain tasks. If vulnerable software gets attacked, it may lead to problems such as leakage of personal information, abnormal function execution, or abruption of services. Moreover, software bugs cost a lot of money to fix and can lead to losses of millions of dollars and even lives [1]. Some kinds of bugs are simple and harmless; others can be more risky. For example, severe bugs can cause a system to stop working for legitimate users (denial of service) or allow an attacker to inject a malicious program and run it, which can lead to the disclosure of confidential data such as credit card information. Another example that cost people lives is Boeing 737 MAX 8 crashes resulting in about 350 deaths [2]. The main cause was a bug in a new system called Manoeuvring Characteristics Augmentation System (MCAS), which is an anti-stall system to prevent the airplane from falling [3]. This system was difficult for a pilot to control and override it [3].

Software testing has been widely used to reduce the number of bugs in the software. Software testing is a process of evaluating a software application to find whether the specifications and requirement are met and to identify any errors or defects to ensure the quality of the product before final delivery [4]. Researchers have proposed different approaches to support automated software testing. Fuzzing is one of the approaches. Fuzzing was first attempted by Barton Miller in 1988 to fuzz Unix utilities [5]. Since that time, fuzzing has become an interesting topic.

Fuzzing is an automated technique that supports discovering vulnerabilities and weaknesses in a target program by using malformed input data from files, network

protocol, etc. [6]. The idea behind fuzzing is generating a large number of invalid or bad inputs and feeding them to the target program to cause a crash or trigger errors. In general, a fuzz testing consists of user inputs (seeds), a target program, fuzzing techniques that use the seeds to generate new malformed inputs, and abnormal behavior monitor. First, user input is an important component of the fuzzing process that serves as the route of discovery for bugs or flaws in a software/system. Second, the target program is the software under test that is needed to conduct fuzzing process on. Third, a fuzzing technique is a strategy for creating fuzzing input files which are malformed inputs that could trigger vulnerabilities in the target program. In the end, the abnormal behavior monitor waits for any exception wheb running the target program with the malformed inputs.

1.1 Software Vulnerability

There are many definitions or descriptions for "vulnerability". In NIST Glossary of Key Information Security Terms, the definition of vulnerability is "Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source" [7]. Some authors have defined it based on their own viewpoints of access control, state space, trust, etc. For example, Shirey defines "vulnerability" as a defect or a weakness in a software/system design, implementation or operation management which can be used to break or violate a security policy [8]. Ivan Krsul states that software vulnerability is an instance of an error, a mistake, or a fault in the software specification, requirements engineering, development, or configuration so the software's execution can violate or exploit the security policy [9].

Software applications may have different kinds of vulnerabilities that may cause security breaches and information leaks which affect the privacy and availability of a

system. Buffer overflow is a common vulnerability in a system, which may impact the system availability and information disclosure. Some common vulnerability types are summarized and explained in the next section.

Table 1.1: Top 10 Vulnerabilities

No.	Vulnerability	Explanation
1	CWE-122: Heap-based Buffer Overflow	A heap buffer overflow is when data overwrite outside an allocated space in the heap section in the memory.
2	CWE-121: Stack-based buffer overflow	A stack buffer overflow happens when data overwrite and exceed the limited space allocated in the stack section in the memory. This vulnerability causes a crash in a program when the data exceeds the stack size.
3	CWE-787: Out of bounds write	Out of bounds write vulnerability is when a program writes data that exceed the size of a buffer or fall behind the beginning of that buffer.
4	CWE-120: Buffer copy without checking the size of input (classic buffer overflow)	Buffer overflow happens when a program takes an input copy of a buffer to an output buffer, and the input buffer size is larger than the output buffer size.
5	CWE-190: Integer overflow or wraparound	Integer overflow happens when a program performs an integer calculation resulting in a value that is either too large or small to store it in the associated representation.
6	CWE-119: Improper restriction of operations within the bounds of memory buffer	A program performs operation on a memory buffer; however, it can read or write to a memory place that is outside the intended boundary of the buffer.
7	CWE-125: Out-of-bounds Read	When a software reads data and exceeds or falls beyond the intended boundary.
8	CWE-129: Improper validation of array index	It is when a program uses untrusted input when doing calculation or using array index; however, the program does not confirm or wrongly confirms the index to make sure the index points to a valid place in the array. The data type that may trigger this kind of vulnerability is integer.
9	CWE-131: Incorrect calculation of buffer size	It is when a program calculates the buffer size wrongly to be used when allocating a buffer, which may lead to a buffer overflow.
10	CWE-134: Use of externally-controlled format string	It is when a program uses a function that accepts a format string as an argument but the format string creates from an external source.

1.1.1 Common Software Weaknesses

Table 1.1 shows the top 10 vulnerabilities in software, according to CWE [10]. CWE is a website which provides information on software vulnerabilities in computer systems.

It is widely used by the computer security community as a guideline for vulnerability description, elimination, and deterrence. CWE started in 2005 and developed as a list of software weaknesses which are more adequate for assessing software security. It lists different kinds of vulnerabilities and also provides examples and mitigation techniques to the weaknesses. Moreover, CWE supports the evaluation of security tools and services designed for discovering vulnerabilities in programs [10].

1.2 Software Testing

Software testing is the process of running a program to check whether the target software meets the specification requirements. Software testing is an important step in the software development life cycle. It is widely used to test the quality of software and reveal bugs in it. Most testing can be done automatically, which reduces the workload of developers and saves time and money as well. Bugs may be triggered during the software testing process and therefore can be fixed before the software is publicly available on the market. Good and effective software testing can reduce the number of vulnerabilities in the software and also make sure the implementation follows the design and the software runs as expected.

Software testing strategies and approaches available in use today can be divided into three main categories, functional testing and non-functional testing [11].

In functional testing, different tests are performed in ordered steps that will lead to successful software testing. The steps are 1) unit testing, which tests a basic and small piece of code; 2) integration testing, which integrates the units to each other to build the software; 3) system testing, which tests the system as a whole; and 4) acceptance testing, which ensures the system is running as intended and the delivery is done [11].

Non-functional testing is a type of software testing that is concerned with the

operational aspects of the software product such as performance, security, usability, compatibility, etc. [12]. Non-functional testing has many types; however, the most common ones include: 1) performance testing, which concerns how a software will act under different test cases; 2) security testing, which provides data and operation protection to ensure integrity, confidentiality and availability of a system; 3) usability testing, which evaluates system's ease-of-use, and 4) compatibility testing, which ensures the system is going to run and work in various environments.

Testing can be categorized to black-box, white-box, and grey-box testing. Both functional and non-functional testing can use them. Black-box testing processes the software as a black box which investigates the software at run time without any knowledge of the internal behavior. White-box testing is the opposite of black-box testing; it knows the internal functionality, logic, and the structure of source code. Grey-box testing stands between black and white box; it allows a tester to examine software while having a partial knowledge of the source code [11].

1.3 Fuzz Testing

Fuzz testing or fuzzing is a type of software testing. Fuzzing is an automated technique that supports discovering vulnerabilities and weaknesses in a target program by using random generated malformed input data [6]. Fuzzing has shown effectiveness in software security; specifically, it helps find flaws in an application before final delivery. Fuzzing can be automated and does not require access to the source code compared to manually reviewing the source code, which requires a huge amount of time and cost. It can trigger vulnerabilities that the programmers overlooked while programming, such as buffer overflow, off by one error, etc. Fuzzing was first implemented in 1988 when Miller Barton attempted to test Unix utilities automatically with random input files. Miller's aim was

to fuzz Unix programs to test the reliability of the Unix system [5]. In recent years, fuzzing has become an interesting topic in software testing. There are many fuzzing tools that automatically generate invalid test inputs and feed them to the target program for exploring any bugs; some known fuzzing tools include AFL [13], Peach [14], LibFuzzer [15], and RADAMSA [16]. There are different types of fuzzers, such as black-box fuzzing, coverage-guided fuzzing, symbolic execution-guided fuzzing, dynamic taint analysis-guided fuzzing, grammar-based fuzzing, etc. Black-box fuzzing randomly generates a stream of test cases and fuzzes a program without any knowledge of the program. Coverage-guided fuzzing uses trivial program coverage feedback to follow how the running path of the program switches via given fuzzed input. The fuzzer employs the collected instrumentation data to choose which inputs could be ignored or kept in the corpus queue. Fuzzers that use symbolic execution technique employ input values as symbolic values rather than using actual values and use symbolic presentation to express the values of program variables [17]. Dynamic taint analysis is a type of data flow analysis method which is used in many domains like software engineering, and computer security. It is used in fuzzing to track the generation of some kind of inputs to collect useful data to fuzz programs with different inputs. Grammar-based fuzzing takes grammars for a certain input file structure such as HTML, XML, C language, etc. to generate valid input fuzzing files that are accepted by the grammars. A detailed literature review on the existing grammar-based fuzzers can be found in Chapter 2.

1.4 DARPA Cyber Grand Challenge Dataset

DARPA Cyber Grand Challenge (CGC) was a competition developed to spur the research on automated vulnerability discovery and patching. CGC representatives created a dataset containing 250 vulnerable stand along programs with detailed documentation of

the vulnerabilities. DARPA designed the DARPA Experimental Cybersecurity Research Evaluation Environment (DECREE) which is a simple operating system and an open source Linux extension made for managed software security trails. DECREE OS contains 7 system calls: `transmit` to send data, `receive` to receive data, `waitfd` to wait for data over file descriptors, `random` to generate random data, `allocate, deallocate` for memory control and `terminate` to stop the server communication [18]. There are about 250 programs in the DARPA CGC dataset. Each program has at least one vulnerability in it. The programs are written in C or C++ programming languages. Despite the simplicity of the CGC environment model, the programs given by DARPA have a large scope of complexity [18].

Although source code for each of the programs is not given during the competition, most of the programs came with network traffic files saved in pcap (Package CAPture) files. These pcap files record the interactions with the programs, what was sent to the program and what the program responded to. The network traffic can be extracted from the pcap files for further analysis and learning.

1.5 Motivation and Objectives

1.5.1 Motivation

Generally, fuzzing is a prevalent and effective way to reveal bugs in applications. Fuzzing tools work by providing a huge amount of fuzzing inputs which can be used to test the target program. By looking closely at the execution of these fuzzing inputs, fuzzing tools can identify inputs which can trigger an exception. In high-level view, one can take into consideration that fuzzing is a random method to discover flaws in software; however, most of the randomly generated inputs are rejected in the early stage by the target program without visiting interesting locations in the program. There are many studies

conducted on this approach to explore new effective ways to generate interesting fuzzing inputs which are more likely to trigger a vulnerability located deeply in a program [19]. Although great progress has been accomplished in fuzzing, there is often still human intervention in the fuzzing process [20]. It is an important goal for fuzzing tool developers to reduce the human interaction and domain knowledge of the target program.

A very important aspect of fuzzing is input generation, which affects the effectiveness of fuzzing input data. Without the source code and knowledge about the program, the performance of a fuzzer will be limited because most of the programs require highly structured input. To help improve performance, some fuzzers require valid sample input to at least have a good start point, but they cannot guarantee that a generated fuzzing inputs meet the program requirement and can be accepted by the program. Some fuzzers limit the format of the target program to be a certain type of program, such as pdf, png, html. Some researchers focus on generating fuzzing inputs based on one or more input formats [19, 21–23]. For example, fuzzer GramFuzz [22] deals with JavaScript, HTML, and CSS formats, so the fuzzer understands how to generate fuzzing input that satisfies the format requirement for those three formats. Several fuzzers require users to manually create grammars to help generate fuzzing input [21, 24–26], but they require the users to have knowledge about the input format and be able to provide the proper grammar to the fuzzer correctly. Therefore, it is still an open question to automatically generate fuzzing inputs which are good for general inputs' format specification (often called "grammars").

Some researchers have studied how effective user inputs can be generated [27]. Others have concentrated on fuzzing techniques that exercise fewer resources and reduce the space of potential inputs [24]. The focus of this research is learning grammar from the sample input files and generating effective fuzzing inputs from grammars. In this research,

the sample input files will be studied and grammars will be extracted from the input automatically. The grammar is later used to help generate effective fuzzing input which has correct format that can be accepted by the program and can go deeper into the program.

1.5.2 The Objectives

The goal of my research is to develop a grammar-based fuzzing tool which can learn the grammar from sample input file. Then the grammar is used to generate effective fuzzing input which can reach deeper into the tested program. The tool was implemented using Python programming language. Particularly, the research has the following objectives:

- Objective 1. Review recent literature on grammar-based fuzzing. Many of the different techniques have been introduced and developed. Having a high-level view of the recent grammar-based fuzzers with their strengths and weaknesses is significant for the research.
    - Task 1.1 Review existing grammar-based fuzzing tools.
    - Task 1.2 Compare the different grammar-based fuzzing tools.
    - Deliverables: A literature review paper has been published [28].
- Objective 2. Analyze input files from DARPA CGC dataset to understand input format and extract grammars.
    - Task 2.1 Collect the data from DARPA CGC environment.
    - Task 2.2 Develop a method to extract grammar from sample input files.
    - Task 2.3 Implement a fuzzer wihich can extract grammar from sample input files.

    * Deliverables: Developed a tool to gather data and generate grammars.

- Objective 3. Employ the extracted grammars to modify the sample input to produce fuzzing inputs.

    * Task 3.1 Develop a fuzzer which can take the extracted grammar.

    * Task 3.2 Use the grammar to help modify the existing sample input to generate fuzzing input files.

    * Task 3.3 Evalaute the fuzzer with DARPA CGC dataset.

    * Deliverables: Developed a fuzzer which can take the grammar and generate fuzzing files. A journal paper has been published [29].

- Objective 4. Learn the order of the commands and utilize the grammars to create new fuzzing inputs.

    * Task 4.1 Learn the structures of the sample input files.

    * Task 4.2 Use the grammars and structure information to generate new fuzzing input files (without modifying the existing sample files).

    * Task 4.3 Test programs with the newly produced fuzzing inputs.

    * Deliverables: An updated fuzzer with the ability to generate fuzzing files without modifying the available sample input files has been developed. A conference paper has been published [30].

- Objective 5. Evaluate the new fuzzing tool and compare it with other fuzzers.

    * Task 5.1 Evaluate the fuzzer with DARPA CGC dataset.

    * Task 5.2 Run other fuzzers with DARPA CGC dataset.

* Task 5.3 Compare the performance of the new fuzzer with other fuzzers.

* Deliverables: Testing results are documented in this dissertation.

1.5.3 Contributions

Most previous research studies centered on generating fuzzing files on a specific input language and using public grammar rules to generate fuzzing input files. The developed tool can analyze sample input files, obtain the grammar rules, and use them to generate fuzzing inputs automatically. This work is a grammar-based fuzzing tool that analyzes sample input files of a program and uses fuzzing techniques to generate effective fuzzing inputs.

The developed tool was designed and implemented based on grammars and effective fuzzing techniques that will improve the efficiency of vulnerabilities detection. Without any human interaction, by using sample input files, the tool can automatically learn the grammar of the input and generate effective fuzzing inputs which may trigger vulnerabilities in a target program. This will provide software developers a hand to test their software, discover vulnerabilities, and make sure their software is safe and secure. The extracted grammar will guide the fuzzer to generate fuzzing files by modifying the sample input files. Therefore, the tool will support software developers and industry to find bugs in a system. Since the tool is automated, it will save the developers and testers time and energy.

The research learns the structure (the orders of the commands, probability of the command following another) of the sample input file as well. This will make the fuzzer able to generate completely new fuzzing files without mutating or modifying the sample input files. It will give conceptual methods based on general programs' inputs grammar analysis and understanding ways to use them to generate new fuzzing input from those

grammars to explore bugs easily. To our knowledge, this will be the first grammar-based fuzzer which can generate fuzzing files based on the analysis and learning of the sample input files. Therefore the research work will bring in new conceptual methods to the software testing community.

1.6 Dissertation Overview

The remainder of this dissertation is organized as follows. Chapter 2 provides background on the fuzzing approach and surveys various grammar-based fuzzing techniques which have been proposed by other researchers and developers. Chapter 3 provides detailed approach about analyzing input files to extract grammar from them. Chapter 4 presents the detailed design of generating fuzzing files by modifying the sample input files with the help of the extracted grammar. Chapter 5 presents the using of Markov chain model to help further analyze and learn from the sample input files. This chapter also discusses the updated fuzzer which can generate completely new fuzzing files. Chapter 6 explains the experiments conducted on the tool and explains the findings. Chapter 7 provides conclusion and future work.

Chapter 2: Background

2.1 Grammar Theory

A grammar is a set of rules which define a valid language [31]. So, a grammar consists of 4 tuples: $G = \{V, \sum, P, S\}$

V: Set of non-terminals

$\sum$: Set of terminals

P: Set of production rules

S: Start symbol

Non-terminal symbols are those symbols which can be substituted as many times as needed by a production rule. Terminal symbols are those symbols which cannot be substituted further [31].

A language is generated from the rules of a grammar. A language contains only terminal symbols ϵ which means null (empty) alphabet or $\sum$. When a language L is generated by a grammar G, then the language is written as L(G) and read as the language L generated by grammar G. The grammar is called G(L) and read as the grammar for language L [31].

The production rules for a grammar consist of two parts: left hand side (LHS), which always contains non-terminal (V) to be substituted, and right hand side (RHS), which may contain non-terminal (V), terminal ($\sum$), or any combination of both. The grammar always starts with start symbol (S) and at least one production rule must have the start symbol (S) in the LHS [31].

For example, if a grammar consists of the following:

$G = \{(S,A,B),(S, A, B, a, b, \epsilon), P, S\}$

P:

$S \rightarrow A \mid \epsilon$

A → aSB

B → b

The generated language will be L = {‘ ‘, ‘ab’, ‘aabb’, ‘aaaabbbb’, ‘aaaaaaabbbbbbb’, . . . }. The language L generated by the grammar G is the same number of ‘a’ and ‘b’ or null.

2.2 Fuzzing Approaches

This section introduces available fuzzing approaches and techniques. It provides an explanation of each approach and advantages and disadvantages of each one.

2.2.1 Blind or Black-box Fuzzing

Blind or black-box fuzzing randomly generates a stream of test cases and fuzzes a program without any knowledge of the program's code. If a fuzzing tool generates fuzzing input without considering the internal actions of a program; it is called "blind fuzzing." Some examples of blind fuzzers include Peach [14], RADAMSA [16], and Sulley [26].

The advantage of blind fuzzers is that they are easy to set up and use. However, one problem with blind fuzzing is that the random input cannot trigger vulnerabilities located deep in the program. It can only find bugs in the surface of the program, because in most of the cases, random input does not meet the format requirement of the program and therefore will be ignored and terminated by the program at the early stages. Therefore, it is difficult for a black-box or blind fuzzer to generate fuzzing inputs that explore a large number of paths in the target program [32].

To make blind fuzzers more effective, most of them require information about the format of the target program. For example, pdf fuzzers know the format of pdf files and therefore can generate fuzzing files which are acceptable by a pdf program. Similarly, there are HTML fuzzers and XML fuzzers that target HTML and XML separately. A blind fuzzer usually uses the format specification to generate new inputs in fairly effective

way. Another way to help blind fuzzers is the use of sample input. A group of valid inputs (corpus) could be essential to aid the generation process [33] [23]. A fuzzing tool can utilize mutation techniques to mutate the initial seed of inputs and generate fuzzing files from there. Additional test inputs are generated from some defined mutational operators randomly on the initial corpus or new test inputs that are created throughout fuzzing execution runs. Some common mutation operations are bit flipping, splicing (recombining two inputs) and repetitions.

Peach [14] is a famous general purpose fuzzing tool that builds a huge amounts of invalid samples based on input format through the component called Peach Pits. Peach generates invalid test cases based on mutation strategies which are the rules for fuzzing system to process the input data. These rules decide which mutation operator will be used to manipulate the input and then generate different number of input test cases. Peach is time consuming because it builds Pit file using input format given by Peach to describe the target file format [32].

RADAMSA [16] is a general purpose fuzzing tool. It is a black-box fuzzer which depends on a sequence of input cases and is not completely arbitrary. It employs the format of valid inputs by looping over the patterns it finds and then fuzzes them based on flipping bits and generates a large number of input test cases.

Sulley [26] is a tool which works by defining test input cases for fuzzing and then describing the target software and running the fuzz data produced from the main mutation method then sends data test cases to the target program for testing. In the end, Sulley provides the result analysis via web-based user interface.

2.2.2 Coverage-guided Fuzzing

A coverage-guided fuzzer uses trivial program coverage feedback to follow how the running path of the program switches via given fuzzed input. The fuzzer employs the collected instrumentation data to choose which inputs could be ignored or kept in the corpus queue. Thus, the fuzzer is able to develop inputs that are different from initial seed while testing new software attributes. This approach offers a fuzzing tool the ability to continuously discover the internal code of the software as it reveals novel paths. Some of the most famous fuzzing tools that employ this approach are AFL [13], T-Fuzz [34], and CollAFL [35]. However, a good coverage does not guarantee the fuzzer can trigger bugs in the software.

American Fuzzy Lop (AFL) [13] is a general purpose coverage-guided fuzzing tool. AFL obtains coverage information at the time of fuzzing operation. It gets the coverage information by a code instrumentation. AFL supports two modes of instrumentation: compile time instrumentation for open source applications, and instrumentation at runtime using QEMU for binaries instrumentation. In compile time instrumentation, AFL supports both GCC and LLVM mode. In runtime instrumentation, AFL supports the QEMU mode that instrument piece of a program.

Peng et al. presented a coverage-guided fuzzing tool, T-Fuzz [34], which is a transformational fuzzer that focuses on improving the fuzzing technique to find vulnerabilities in a target program by removing complex and hard checks. It utilizes AFL fuzzing tool to fuzz the target program. When the fuzzing tool is unable to generate new inputs to find bugs, a tracing approach discovers all the input checks generated by fuzzing tool that fail to satisfy the checks. Then, the program is transformed by disabling or removing the hard checks. After that, the transformed program continues to be fuzzed independently

by AFL. The authors determined that T-Fuzz is able to find bugs because the T-Fuzz found 166 out of 296 in CGC dataset (more than AFL and Driller had found). Also, T-Fuzz showed better performance in LAVA-M dataset.

Gan el at. [35] proposed a code coverage-guided fuzzing tool which is based on coverage sensitive fuzzer called CollAFL. The purpose of CollAFL is to solve the hash collision problem in AFL. By verifying that each edge in the program has its own hash, it helps AFL to distinguish between two edges. Then, the fuzzer can give accurate edge coverage information. CollAFL uses the coverage information to apply three new strategies to increase the speed to explore new paths and bugs. They proved that CollAFL is successful in finding new interesting paths that can crash the program and discover vulnerabilities.

2.2.3 Fuzzing Guided by Symbolic Execution

The idea behind symbolic execution technique is to use input values as symbolic values rather than using actual values and to use symbolic presentation to express the values of program variables [17]. Symbolic execution plays an important role in software testing because it helps discover a large number of program paths [36]. Symbolic execution can efficiently generate high code coverage test cases and find bugs deep in a complicated software because it has the capability to generate real test inputs [36]. This approach overcomes the problems in coverage-guided fuzzing by directing fuzzing tool to fuzz the interesting locations in the software. However, this approach has two main issues: path explosion and generating complicated constraints [36]. For example, if software has a complicated iteration or function that operates recursively, there probably would be an infinite number of paths [36]; then the constraints could be very complicated and might not be resolved in a timely manner [36]. Some fuzzing tools that employ symbolic execution are Mayhem [37], Driller [38], EXE [39], and KLEE [40].

Mayhem [37] is a cyber reasoning tool that uses symbolic execution approach. Mayhem is a system that is developed to automatically discover bugs in a software. It works by inserting random initial input samples which Mayhem uses to trace the concrete execution of the target program, while building a symbolic procedure extracted from the input. Then, when Mayhem finds a branch condition, it utilizes the procedures for the condition to create a malformed input test case that takes the other direction of the branch. Then, Mayhem repeats the tracing process using the input test cases as a new corpus. After repeating, Mayhem enlarges the corpus to cover a larger number of paths. Mayhem reduces the path explosion problem by combining similar corpus together and then processing a single merged corpus that accomplishes same discovering to process each of its component corpus.

Driller, introduced by Stephens et al. [38], is a hybrid vulnerability finding tool which employs selective concolic execution method to enhance the efficiency of fuzzing and find deeper bugs. When fuzzing cannot identify inputs that direct to new interesting paths, the concolic execution system will be invoked. All of the interesting inputs that were found by the fuzzing tool will be passed to the engine. After that, the engine is going to recognize state transitions and create input that could cause execution to the particular state transition. The input information will be sent back to the fuzzing tool to explore the newly found compartment. Driller is created on top of AFL [13]. Stephens et al. combined the fuzzer AFL with their concolic execution system, so Driller takes advantage of the speed up of fuzzing with the selective concolic execution technique to find deep bugs.

EXE (EXecution generated Execution)is another symbolic execution fuzzer that was introduced by Cadar et al. [39]. EXE dominates other traditional run time tools as stated

by the authors; because it has the ability to force running down any path in the program, and detect if the current path constraints accept any value which leads to a flaw or not when running a severe vulnerability. When a path exits or finds a flaw, a constraint solver helps to resolve the path restrictions to explore actual values, then EXE can automatically produce an input test that will execute this path.

KLEE [40] is an extended work of EXE and is also a symbolic execution tool. Cadar et al. stated that KLEE has the ability to create input tests that reach high code coverage on various sets of complicated software. It uses different kind of constraint-solving improvements and utilizes search heuristics to give high code coverage. Moreover, it can evaluate weakness constraints to capture buffer overflow vulnerabilities. KLEE at the start analyzes the code to gather data about arrays and their size; after that, KLEE employs these data to compute the weakness constraints. KLEE takes advantage of applying both weakness constraints and path constraints to effectively generate fuzzing input file.

2.2.4 Fuzzing Guided by Dynamic Taint Analysis

Dynamic taint analysis marks data found or derived from untrusted sources as tainted [41]. Dynamic taint analysis is a type of data flow analysis method which is used in many domains like software engineering and computer security. The idea behind it is to trace the propagation of a specific part of data such as password or text data, then gather some good and helpful information from the executed program [42]. Dynamic tainting analysis approach has been commonly employed to stop exploitation of vulnerabilities such as buffer overflow and control format string [43]. The benefit of dynamic taint analysis is the ability to discover many input validation attacks with a low false positive rate; however, it is time consuming, which means the execution of dynamic taint analysis is slow and

misses full code coverage paths [44]. Some fuzzing tools that use dynamic taint analysis are BuzzFuzz [45] ,TaintScope [46], and Dowser [47].

BuzzFuzz [45] is a white-box fuzzing tool which employs dynamic taint analysis to find initial inputs that affect the defined values applied on the attack point of the application. After that, it produces new fuzzing test data that only fuzz the known input cases. Rather than utilizing random fuzzed input test cases, it employs the direct values that are related to the particular possible weaknesses. The advantage of BuzzFuzz is that it is able to keep the valid input file format, so they can meet the basic syntactic restrictions and reach deeper locations in the application.

TaintScope, introduced by Wang et al. [46], is a fuzzing tool that combines taint analysis and symbolic execution. It employs dynamic taint analysis to continue tracing the fields of input data that affect the security processes. These input fields are known as hot bytes. After that, TaintScope produces the fuzz input data with random or edge values based on hot bytes. Additionally, the tool employs symbolic execution approach because it reacts with hot bytes in initial input as symbolic values and uses dynamic symbolic execution to reveal bugs and weaknesses on the track.

Dowser [47] is also a fuzzing tool that integrates dynamic taint analysis and symbolic execution. Dowser employs static analysis to find critical locations in the application which may cause buffer overflow. After that, it applies taint tracing method to locate the input source which are operated by these identified locations. After that, a new path guidance function is going to direct the symbolic execution to discover these critical locations.

2.2.5 Grammar-guided Fuzzing

Grammar-guided fuzzing approach is suitable for structured input specification like HTML or CSS. Blind fuzzing tools have hard time handling these kind of input files because they possibly break critical features of the input that the software is going to rapidly detect in the beginning of parsing stage and then ignore. Therefore, blind fuzzing tools end up testing a small portion of the software code [27]. Grammar-guided fuzzing is a strong approach that takes advantage of grammar and supports fuzzing tools to go deeper in the application. Therefore, blind fuzzing tools can employ this approach to increase their efficiency to discover bugs in the target program. However, it is dependent mostly on the provided grammar, which requires human efforts to write the grammars and is time consuming and error prone. There are some fuzzing tools that use this approach, such as previously mentioned tool Peach [14], Jsfunfuzz [48], and LangFuzz [49].

Jsfunfuzz [48] is a grammar-guided blind fuzzing tool for JavaScript web browser engines which is written in JavaScript too. Jsfunfuzz focuses on confirmed classes of bugs. It discovers dangerous operations like crash or hangs and finds bugs and errors by differential testing. It is a generative tool which uses a provided input grammar. It begins with the start symbol and picks production rules randomly. Then, it replaces the non-terminals in a form of string that is defined in terminals. Eventually, it generates too many input strings. The tool found more than 1000 bugs in Mozilla JavaScript engine and was faster in finding these bugs than other tools. It has become a common tool for web browser designers.

Holler et al. [49] introduced LangFuzz, which is a grammar-guided fuzzing tool that employs grammars for multiple languages and uses seed samples. LangFuzz builds knowledge from the target language to identify variables and existing keywords. After

that, the tool uses mutation technique to modify the inputs which then generates test cases to examine the target program.

2.3 Literature Review on Grammar-based Fuzzing

Grammar-based fuzzing is a fuzzing technique that takes a particular input format to get the correct grammars structure. It uses the grammars to generate fuzzing input that passes the parsing stage of a program. The vast majority of the grammar-guided fuzzing techniques take unique file format for specific input structure. A fuzzing tool will have difficulties if it does not have a valid user input format known before. Therefore, it is critical for grammar-based fuzzing tool to have grammar specification which will support the generation of valid fuzzing inputs that meet the program testing and assist in exploring interesting bugs. Moreover, grammar-based fuzzing tool that uses grammar guided method is able to increase code coverage and reach deeper locations in a target application [50]. Most of the techniques that use grammar-based fuzzing utilize grammars with user inputs in the beginning of fuzzing process to generate test. By using grammars, the fuzzing tool can generate new user inputs and then apply them to generate new fuzzing inputs. Many studies have been performed on fuzzing techniques that use grammar to guide fuzzer. Moreover, they combine grammar-based fuzzers with techniques such as mutation, machine learning (e.g. neural networks), evolutionary computing (e.g. genetic algorithm), or coverage feedback to guide fuzzing and improve the ability of revealing bugs in a program under test.

Mutation-based fuzzing takes sample inputs and chooses them in particular order, then mutates/changes them in different ways, and examines target programs with the newly generated test input. Mutation is the most common technique used for fuzzing guidance because it is an efficient way to get fuzzing user input that supports finding deep

bugs while using it with grammar-guided fuzzing [22]. Machine learning fuzzing is another method that a fuzzer can use to find bugs in the target programs. It is a learning algorithm which learns or trains model to do certain operations with some probabilities, and it can be used with fuzzing to generate intelligent or organized fuzzing files. Evolutionary computing fuzzing is inspired by evolution theory and generates new individuals in the eco-system with reproduction and combination of good features of individuals by using fitness function. Therefore, the tool can generate effective fuzzing inputs. As mentioned before, coverage-guided fuzzing is a technique in which a fuzzer can get coverage feedback information so the fuzzer uses it to test unvisited locations in the program under test.

2.3.1 Grammar-based Fuzzers Based on Mutation

Some studies used mutation techniques in a grammar-based fuzzing. According to Guo et al. [22], GramFuzz uses the two techniques of grammar analysis of inputs and mutation of the input structure to fuzz web browsers. GramFuzz obtains initial input file from the internet and analyzes them to set a parse tree by using Gold Parser, an open source analysis tool. After getting the grammar trees, the nodes are mutated. Then input test cases can be generated. Authors reported that by combining generation and mutation, test cases will be more effective at fuzzing web browsers. GramFuzz has found 36 vulnerabilities considered severe security in IE and Mozila [22]. However, GramFuzz only works with web browsers which accept HTML, CSS, and JavaScript files.

Sargsyan et al. [51] presented SD-Gen which is a structure input generation grammar-based fuzzing by using available grammars in ANTLR that supports grammar rules for more than 120 languages and file formats. It takes the target grammar and input language as inputs. Then, test data (programs) are generated and mutated. SD-Gen can generate programs for compilers, interpreters, and translator testing because it supports generating

programs in C, C++, Java, Python, etc. Results showed SD-Gen is able to increase code coverage [51]. However, it can't provide programs semantic correctness.

BlendFuzz [52] is another grammar-based fuzzing tool which is a model-based framework that is effective fuzzing with grammatical inputs. BlendFuzz generates inputs by first building a parser for the target language. Second, it applies the parser to seed set (input) and obtains a parser tree to extract grammar language from the tree structure and check the ordering of grammar elements. Third, it maps between inferring grammar elements and collecting language constructs to make it easy for mutation. Lastly, it uses mutation technique to the string by selecting fragments and replacing them with the same type. These inputs test the target program so the complex structures and program's edges are covered. In the end, the results showed the approach is effective and enhanced the code coverage and revealed security vulnerabilities [52]. However, it can't generate new grammar components other than those available in the seed set and can't always generate syntactically correct inputs. Also, it supports only specific input formats XML, HTML, and Javascript.

QuickFuzz [53] has been introduced by Grieco et al. It leverages Haskell's QuickChick (the well-known property-based random testing library) and Hackage (the community Haskell software repository) and combines a general purpose bit-level mutational fuzzers (e.g. Radamsa and HonggFuzz) to create fuzzing automatically for some well-known file format without providing any file format specifications. It generates input files based on a grammar then performs mutational techniques on these files to trigger unexpected behavior in the target program. QuickFuzz was tried with real world programs and found to be successfully effective in discovering most important vulnerabilities [53]. However, at the start of the generation process, some randomly derived inputs are not effective in

generation of source code because they are rejected in the parsing stage. Also, there are some issues of using third party's package because some modules do not support certain complex file types. The fuzzer mostly focuses on multimedia file formats such as image, audio, video, etc. and supports certain PDF and certain types of archive file formats such as ZIP and TAR.

LangFuzz [49] was introduced by Holler et al. It is a fuzzing tool using blackbox fuzz testing of engines based on grammar. LangFuzz takes the provided grammars using ANTLR and sample code to generate language fragments and test suite for code mutation. Code mutation is divided into two phases: a learning phase and main mutation phase. In learning phase, a group of sample codes are operated with a parser using grammar. The parser will separate the input code sample into code fragments which are non-terminal in the grammar. Once the learning phase is finished, mutation phase starts by selecting some code fragments and replacing them with others of the same type. Using code generation, step wise expansion is used by considering code as syntax tree. LangFuzz uses code mutation, which is the primary technique, and random generation to generate test cases to test the engine before passing the test cases to the interpreter. By combining two types of code generation (mutation and generation), LangFuzz has found 164 real-world bugs in popular JavaScript engines and 31 security related vulnerabilities from Mozila and Chrome V8 and detected 20 bugs on PHP engine [49]. LangFuzz has some issues; for example, fewer test cases or biased tests decrease LangFuzz's performance. Moreover, it has to make necessary changes to add grammar rules to use the tool for generating new inputs to be compatible and accepted by the new language. Also, it only supports testing Mozilla and PHP interpreter.

2.3.2 Grammar-based Fuzzers Guided by Machine Learning

Some grammar-based fuzzing tools utilize machine learning to generate well-formed inputs that are able to increase code coverage and discover new bugs.

Godefroid et al. [54] designed Learn&Fuzz, which uses machine learning (Neural Network) to generate fuzzing inputs automatically. Learn&Fuzz uses neural network-based learning methods to learn a grammar for non-binary PDF data objects such as formatted text. It uses input sampling techniques to generate PDF objects from the learned distribution. The full specification of the PDF format is over 1300 pages long. The grammar rules are huge and heavy but they are structured well and adequate for learning with neural network. Learn&Fuzz utilizes learned input probability distribution to guide the tool to generate fuzzing inputs in a smart way. It shows that the neural network technique is able to generate well-formed inputs and increase coverage of input parser more than different variations of random fuzzing [54]. Nevertheless, it focuses only on PDF file format. Also, Learn&Fuzz is not able to process less structured inputs such as images, videos, and audio files because it is designed for text formatted inputs.

Wang et al. [19] stated that Skyfire is a data-driven seed generation approach. It takes a corpus and grammar as inputs and generates fuzzing inputs in two steps. First, it parses the collected samples based on the grammar and generates the AST (Abstract Syntax Tree) trees. Then, it learns the PCSG (Probabilistic Context-Sensitive Grammar) that is based on semantics rules and syntax features. Second, it generates seed inputs by looping to select and apply grammar rules that satisfy the context on non-terminal symbols until there is no more non-terminal symbol in the output string. Then, it applies low probability on high probability production rules to obtain uncommon input with variety of grammar structure. Skyfire makes selections on the resulting input seed to

filter out the similar seeds to reduce duplication. In the end, Skyfire mutates the selected input seeds by randomly selecting leaf-level node in the trees with the same node with applying semantic rules and remaining grammar structure. The results show that Skyfire effectively improves the code coverage and enhances the ability to find bugs [19]. However, it is only limited for files whose format are XML, XSL, and JavaScript.

According to Hu et al. [55], GANFuzz uses machine learning for industrial network protocol to fuzz network protocol in which input test is generated using protocol grammar by creating specifications or reverse engineering from network packets. In this study, by using machine learning (neural network) an automated test input is generated with deep-learning techniques to learn protocol grammar. After that, GANFuzz takes advantages to train test inputs over the network packets to get protocol grammars then generates invalid input messages that lead to discover some bugs and errors. The results showed that GANFuzz is effective in code coverage and deeply testing [55]. However, it is only limited for the format of industrial network protocols files. Also, there are some limitations; for example GANFuzz cannot handle file operations or generate correct syntactically protocol specifications, and there are graphical user interface errors.

2.3.3 Grammar-based Fuzzers Guided by Evolutionary Computation

Some grammar-based fuzzing tools use evolutionary computation methods such as genetic algorithm and genetic programming. These tools leverage the crossover and mutation techniques to produce well-formed test inputs based on improved fitness function.

Hodován et al. [56] created the fuzzing tool Grammarinator which works with generation and mutation-based fuzzers with the help of grammar. It uses parser grammar to generate input test cases and build abstract syntax tree for each test case and analyzes them. Moreover, an evolutionary algorithm is used for mutation and recombination to

the test input files. Grammarinator defines the depth of generated structure and focuses its generation on the less visited parts. It also defines complex actions and decides the correct test cases that the grammar can describe. Grammarinator is used to test different JavaScript engines and is found to be useful and effective: it has found more than 100 new issues from IoT.js [57]. However, it only fuzzes JavaScript engines.

Veggalam et al. [23] developed a fuzzing tool called IFuzzer that uses evolutionary computation techniques such as genetic programming to guide fuzzing. it takes context-free grammar as input to generate test cases by generating parse trees and extracting code fragments from test suit. IFuzzer uses genetic programming technique that utilizes mutation and crossover and leverages the improving fitness function to enhance the effectiveness of generating input (codes) test cases. The results showed that IFuzzer reveals bugs more quickly than state-of-the-art fuzzing tools [23]. Moreover, it found 40 bugs in old version of JavaScript interpreter of Mozilla and 17 bugs in latest version of the interpreter [23]. However, IFuzzer is low quality with respect to code generation, and it requires changes for new language or new code.

2.3.4 Grammar-based Fuzzers Guided by Coverage Feedback

As stated previously, latest mutational blind (black-box), coverage-guided, symbolic execution, or dynamic taint analysis guided fuzzing tools are not able to effectively create inputs for programs with structured input languages. To solve this problem, generational fuzzing tools (whether blind, coverage-guide, symbolic execution, or taints analysis) use a specification or a format of the input language (usually called “a grammar”) to generate valid inputs. So, they decrease the space of potential inputs to a small group that has higher probability to trigger vulnerabilities. Moreover, coverage-guided grammar fuzzing tools can mutate inputs in this small group by using the given grammar. We call these

mutations comprehensive mutations since they modify large part of the input. Hence, the performance of fuzzing tools that use this approach can be significantly high by providing formats or specifications of the inputs to the fuzzing tools. Moreover, the fuzzing tool has the ability to reasonably merge inputs that lead to critical attributes with a high probability of discovering more dangerous actions. However, they need human labor and expert knowledge, and they must be manually written to give input format correctly. Also, it is error prone, which is difficult to manually provide a correct specification. There are several tools that employ this technique: they are NAUTILUS [24], AFLSMART [58], and Superion [59].

Aschermann et al. [24] proposed a design and implementation of a fuzzing method that combines the use of grammar to generate inputs with coverage-guided fuzzing to explore bugs deep in a target program that occur after syntactic checks. NAUTILUS does not need a seed input; only needs source code of a target program and a grammar as inputs. Then, it uses the provided grammar for generating new inputs for the program. It employs feedback to generate mutated interesting inputs for the target program. As a result, using feedback with grammar led to a great improvement for fuzzers to find bugs deep in the target program. NAUTILUS has found 13 new bugs in 4 targets Ruby, Lau, PHP and JavaScriptand; and received in a sum of $2600 as award in 6 CVEs that have been discovered [24].

Pham et al. [58] introduced AFLSMART. It is a smart grey-box fuzzing (SGF) and acts as a coverage-guided grammar fuzzing tool. It takes advantages of high-level structural specification of files to generate novel input files for some applications that process complex file formats. New novel mutation operators have been created. They work on the virtual file structure to maintain file validity and discover completely new input files with the help

of SGF validity based power schedule to generate input files. These files are able to pass the parsing part of a target program to increase the likelihood of reaching vulnerabilities deep in the program. It achieves more branch code coverage, up to 87% improvement compared with AFL [13] and finds more flaws. However, AFLSMART is tedious and requires more effort and time to write specification for inputs [58].

Wang et al. [59] proposed a grammar-aware coverage-based gray-box fuzzing tool called Superion. It is used for programs to process structured inputs. It takes as inputs a target program and a grammar of the input tests that are publicly available. They used AST for grammar-ware trimming technique which trims or modifies test inputs and at the same time keeps the input structure valid. The proposed technique removes each subtree in the AST for a test input and watches the coverage changes. Also, the proposed approach includes two grammar-aware strategy; then, a tree-based mutation strategy is proposed that replaces a subtree from itself or another test input in the queue on the subtree in the AST of a test input. It found 34 new bugs in Jerryscript, XML engine, 3 JavaScript engines, and ChakraCore. Also, it discovered 22 new vulnerabilities with 19 CVEs identifier assigned. In comparison with AFL and Jsfunfuzz [48], AFL found only 6 bugs of 34 and Jsfunfuzz did not discover anything [59].

Chapter 3: Input Files Grammars Analysis

After examining the programs in the DARPA CGC dataset, it was determined that each program is a standalone program and there is no similarity among the programs. For example, there are ship game, palindrome test, picture analyzer, file reading and searching programs in the dataset. Each program has its own way to interact with the user, such as using specific commands and special strings. To be able to conduct effective fuzzing, the formats of how to interact with the programs have to be learned. Since most of the vulnerable programs come with network traffic data, the interactions between the user and program can be extracted. This network traffic data can be treated as valid sample input files, and a grammar can be learned from these files and used in the later process.

This section discusses how to analyze input files from the DARPA CGC dataset to extract grammars. After the grammar is extracted from the sample input files, it can be fed to the fuzzer to generate effective fuzzing input files.

This stage begins by collecting data from the DARPA CGC dataset which then starts analyzing for grammars. After investigating the input files/network traffic files in the DARPA CGC dataset, we found that most of the programs require input that are command-like. One example can be found in Listing 3.1, which shows all the messages sent from the user to the program. By taking a closer look, we can find the messages are in the format of command and one or more parameters, command only, or some random strings. Therefore, when analyzing the input, we treat the first word as command and the later part as parameters. For example, in Listing 3.1, commands are the first word of each line, for example, *copy, list, show, erase, make, last, write,* etc.. Parameters are the items that come after a command. Using the first line in Listing 3.1 as an example, *copy* is a command and the parameters are *README.txt*, and *lcggfldeuausqsrx*. On the second line, *list* is a command and no parameter after it.

Listing 3.1: Filesystem_Command_Shell User Input Example

```
1  copy README.txt lcggfldeuausqsrx
2  list
3  show README.txt
4  erase authentication.db
5  make gtims
6  last README.txt 10
7  show lcggfldeuausqsrx
8  show README.txt
9  list
10 write authentication.db
11 show README.txt
12 last README.txt 10
13 last README.txt 10
14 first README.txt 10
15 last lcggfldeuausqsrx 10
16 first authentication.db 10
17 list
18 logout
19 evjfrtm 58932
20 xboskhi 18012
21 ahpoyva 1614
22 last authentication.db 10
23 make flxzzofveqexsfje
24 make kzoycqeppd
25 write flxzzofveqexsfje
26 chwvsfnnspychdjfrbyfjunhvlbypwibjeiuixgoipvxr
27 write authentication.db
28 erase README.txt
29 first lcggfldeuausqsrx 10
30 perms authentication.db 2
31 list
32 make zmhvbqsjmzo
33 perms README.txt 2
34 last authentication.db 10
35 first flxzzofveqexsfje 10
36 perms authentication.db 1
37 erase flxzzofveqexsfje
38 last zmhvbqsjmzo 10
39 perms zmhvbqsjmzo 1
40 perms README.txt 3
41 erase gtims
42 perms authentication.db 4
43 write authentication.db
44 exit
```

The grammar analysis step entails going through all sample input files for a program

and analyzing them to extract the commands with their parameters and use them for generating fuzzing files in the next stage. This step can be further divided into four smaller tasks: 1) read sample input files, 2) analyze real commands, 3) analyze real parameters, and 4) analyze numbers.

3.1 Reading Input Files

The tool starts by reading all the sample input files and saving all of the strings from the input files into an *input list* because it has to be easy and flexible so that the tool can find repeated or real items in all the input files. Then it scans through the input list and counts the numbers of each possible delimiter (, / | : ; = - ‘ space). The reason for this is the tool needs to determine which character should be treated as the delimiter to split the words on each line. From the manual examination of the programs, most programs use white space as the delimiter; however, some programs use hyphen, semicolon, or equal sign to separate command and parameters. Therefore, a possible delimiter list is maintained and the tool first scans and counts the occurrence of each possible delimiter and uses the top one as the delimiter to separate words in the input files. For example, the delimiter for Listing 3.1 is “space” after the tool counts all the possible delimiters in all input files. Then the delimiter is used to split the input strings so that the first word before the first space is treated as command and the later part of the line is treated as parameters of the specific command. The delimiter is used to separate parameters as well.

3.2 Analyzing Real Commands

This stage is for extracting and analyzing real commands. After determining the delimiter, each string in the *input list* is split. Then, each of the first words after splitting the lines will be treated as a possible command. The possible commands are kept in a command list.

```
['ahpoyva1', 'chwvsfnnspychdjfrbyfjunhvlbypwibjeiuixgoipvxr0
 ', 'copy2', 'erase1', 'erase1', 'erase1', 'erase1',
 'evjfrtm1', 'first2', 'first2', 'first2', 'first2',
 'last2', 'last2', 'last2', 'last2', 'last2', 'last2',
 'last2', 'list0', 'list0', 'list0', 'list0', 'logout0',
 'make1', 'make1', 'make1', 'make1', 'perms2', 'perms2',
 'perms2', 'perms2', 'perms2', 'perms2', 'show1', 'show1',
 'show1', 'show1', 'write1', 'write1', 'write1', 'write1',
 'xboskhi1']
```

Figure 3.1: Sorting Commands.

```
{'erase1': 4, 'first2': 4, 'last2': 7, 'list0': 4, 'make1':
 4, 'perms2': 6, 'show1': 4, 'write1': 4}
```

Figure 3.2: Counting Real Commands.

Because of the large amount of input strings, we cannot treat each first word as a command. Instead, only the commands which have higher occurrences should be treated as commands. For example, some commands showed more than 500 times, but some other potential commands only occurred for 2 or 3 times. These low-occurrence commands are less likely to be the real commands, and they should be ignored. Otherwise, the number of real commands will be too high, and it will take much more time to analyze them. In addition, many of the low-occurrence commands are random strings and are not real commands.

To determine if a command is a real command, we count how many times it occurs. If a possible command occurs in more than 10% of the lines, it is considered a "real command". This percentage number was chosen experimentally (see Section 6.6). The reason for doing that is to not include unnecessary items as commands. We set the maximum number of real commands to be 20. So, the tool will not get too many real commands. Therefore, the tool will be able to speed up the process of analyzing larger input files with huge data input and support generating effective fuzzing input files by focusing only on the found items.

Using the input in Listing 3.1 as an example, after the commands are collected, the command list is taken and sorted as shown in Figure 3.1. Then, real commands are collected and the number of occurrences of each one is counted (see Figure 3.2). The reason for sorting and counting commands is to make it easier, more accessible, and faster at finding similarities in commands. After that, the tool will have commands collected with its counter number. Then, the tool keeps those that occur more than 10% of the lines. For example, the final commands list for Listing 3.1 is shown in Listing 3.2.

Listing 3.2: Real Command List

```
1 ['erase1','first2','last2','list0','make1','perms2','show1',
2 'write1']
```

In the command list, an extra number is added after each command to specify the number of parameters.The number is determined based on the number of parameters after each command. By having this number, the tool knows how many parameters come after a command and collects them to find real parameters (next step). Using the first line in Listing 3.1 as an example, *copy README.txt lcggfldeuausqsrx*, the number of the split words on this line is 3, which means there are 2 parameters following the *copy* command. The tool will store *copy2* on the final command list if it exceeds the percentage number so that the tool knows that this command *copy* has 2 parameters. However, if the command doesn't have a parameter, the new command will be command0, such as *list0* in the example. To lower the amount of generated fuzzing files, the max parameter number is set to 3. This means if a line has more than 4 split words, it will be excluded from analysis or fuzzer. After knowing the command and how many parameters it has, each command and its parameters are put together in a general dictionary.

3.3 Analyzing Real Parameters

The goal for the third step is to find common and most available parameters for each command by analyzing all the parameters for that command similar to how we found commands. The tool manages all the parameters and identifies the parameters that appear mostly with a command if the counted number is equal to or exceed a calculated percentage.

Similar to the percentage number used for determining real command, another percentage number (10%) is used. This percentage number can be changed if necessary. The reason for using the percentage number is to limit the number of real parameters and avoid treating random strings as real parameters.

If a real parameter is found, the same process will be done for the second and third parameters. However, if there is no common first parameter, there is no need to find common parameters for the second and third parameters. Similarly, if there is a common first parameter but there is no second common parameter, the tool will not find the third common parameter.

Listing 3.3: Real Commands and Their Real Parameters

```
1  {'copy2': [['README.txt'], ['authentication.db']],
2  'erase1': [['README.txt'], ['authentication.db']],
3  'exit0': [],
4  'first2': [['README.txt','10'], ['authentication.db','10']],
5  'last2': [['README.txt','10'], ['authentication.db','10']],
6  'list0': [],
7  'logout0': [],
8  'make1': [],
9  'perms2':[['README.txt'],['authentication.db']],
10 'show1': [['README.txt'],['authentication.db']],
11 'write1': [['README.txt'],['authentication.db']]}
```

After that, the new findings for real commands with their real parameters are kept for further processing. The final findings from Listing 3.1 are shown in Listing 3.3. On lines

2 and 4, *'erase1'* and *'first2'* are commands. Also, line 2 shows the real first parameters [['README.txt'],['authentication']] for command *'erase1'*, which is a one-parameter command type. The command *'first2'* is a two-parameter command type that has [['README.txt','10'],['authentication.db','10']], which has two first parameters, 'README.txt' and 'authentication.db'. Each one is followed by a second parameter, which is '10' for each one.

3.4 Analyzing Numbers and Change to [0-9]

While manually examining the sample input files of DARPA CGC dataset, it was noticed that some input files, such as "move r1, move r4, move r7, move r10", have different numbers in their parameters. The first parameters for the *move* command are r1, r4, r7, r10. If we compare the occurrence number of each one with the percentage number, then all of them will be dropped. However, they should be treated as a common parameter r[0-9]. Therefore, to be able to detect parameters like this, the numerical parts in parameters are modified and changed to [0-9] to make it easy to detect these parameters. So the first parameters for the *move* command (r1, r4, r7, r10) will be each changed to r[0-9], and they will be counted for an occurrence of 4 and saved in the common parameter list.

After this stage, a final grammar shown in Listing 3.4 will be obtained. The only changes from previous listing (Listing 3.3) are on line 10 in Listing 3.4. The changes can be seen for command ***perms2*** because the second parameters are different numbers for this command. So, all numbers in the second parameters were changed and modified to "[0-9]". Therefore, it was easy and helpful for them to be discovered. "[0-9]" was included with the parameters in the parameters' list.

Listing 3.4: Real Commands and Their Real Parameters after Changing Number to (0-9)

```
{'copy2': [['README.txt'], ['authentication.db']],
'erase1': [['README.txt'], ['authentication.db']],
'exit0': [],
'first2': [['README.txt','10'], ['authentication.db','10']],
'last2': [['README.txt', '10'], ['authentication.db', '10']],
'list0': [],
'logout0': [],
'make1': [],
'perms2':[['README.txt'],['authentication.db'],
['README.txt', '[0-9]'], ['authentication.db', '[0-9]']],
'show1': [['README.txt'],['authentication.db']],
'write1': [['README.txt'],['authentication.db']]}
```

Another reason to change numbers in parameters is that the fuzzing tool will substitute the "[0-9]" to generate random numbers in the hope of discovering any vulnerability caused by using invalid numbers in the program. Some input files have numbers in them, which means there could be a chance of one of those numbers triggering a buffer overflow, integer overflow, or other vulnerabilities.

Analyzing sample input files is started by taking all of input files and going through steps. First, reading the input files which read all input files for a program and find most used delimiter in them. Then, the tool splits them by the delimiter and keeps them for later process. Second, analyzing real commands by taking all commands and count each one and check of the counted number exceeds or equals to to 10% of number of sample input files. Third, analyzing real parameters for each real command and count them to find if the counted number exceeds or equals to 10% of number of command occurrences. Fourth, analyzing numbers in parameters and change them to [0-9].

Chapter 4: Generating Fuzzing Files

After finishing the sample input files analysis, grammars are extracted, which will support generating fuzzing files. This chapter discusses how the new fuzzing tool creates and generates fuzzing files by using the extracted grammar. To be able to test the deeper code in the program, the sample input files are used and modified because we want to keep the orders of commands as shown in the valid sample input files. Also, they were used to help generate effective fuzzing files which can interact with the program correctly. Therefore, the strategies of generating fuzzing files in this step are to make substitution and replacement on the valid sample input files. In this step, a random string generator is developed. The new fuzzer gets a sample input file and figures out which part to substitute with the help of the extracted grammars. Then it uses the random string generator to generate a random string to substitute a particular part of the sample input. This new fuzzing file will be saved and later will be fed into the target program for testing.

The step can be divided into two major steps: 1) Read in sample input file and identify the location in the sample input file where it needs to be substituted, and then construct a new line to substitute the original line, and 2) Generate a random string and substitute the identified part in the sample input file with newly constructed line and save it as a new fuzzing file.

4.1 Line Replacements

The purpose of this step is to replace a line of a sample input file each time with new random string because most programs require keeping the order of the commands in user inputs so the programs run and execute correctly. After having one sample input file, the tool can use it to create multiple fuzzing files from it by substituting different lines.

4.1.1 Constructing Fuzzing Files

In the beginning, the tool starts to generate fuzzing files by opening a sample input file, then all the data from that file will be copied to the new text file. Then, it takes a line from the sample input file and creates a target line variable to recognize the line that needs to be replaced with a random string. Later, it starts replacing lines from first line and then goes to the next line by increasing the target line variable by 1 and continues until it reaches the last line. After replacing a line, it copies the lines before the target line and the following lines that come after it to create a new fuzzing file. In short, it only modifies one line from the sample input file and saves it as a new fuzzing input file. Then, next round, the tool substitutes another line from the same sample input file to make it a new fuzzing file. The tool continues until reaching the last line. Because every time, only one line of the valid sample input is changed, the new fuzzing file is very likely to interact the program properly. By keeping the order of commands in the testing file, the programs can be executed correctly, and the tool will have a higher chance in discovering a vulnerability deeper in the program. When replacing a line, the tool works by taking the first word of it which is considered a command and determine whether it is in the grammar list. Based on the number of parameters, the tool will be able to recognize the type of a command; the command types are a command only, one-parameter command, two-parameter command, and three-parameter command. After that, when the tool knows the format of the line, it is compared with the extracted grammars. If it is one of the commands, then it continues to parameter checking. However, if the first word does not match any command in the real command list, then the tool will generate a random string for that line, supporting testing programs that do not have real commands.

4.1.2 Creating Fuzzing Line

In a line, the tool will have different number of parameters that come with a command. There are commands with/without parameters. For each one, the line is taken based on if it is a command with zero, one, two, or three parameters. Then, any number that is found in a parameter is changed to "[0-9]" for each parameter. After that, it is compared with the grammar extracted before. If there is a match, the numeric parameter will be recognized, and it will be substituted by a random number with a random length. However, if there is no match, the tool will consider these parameters as strings. So, for a command with no parameter, the tool can substitute the command and replace it by a random string with different length or keep it with no replacement. For a command with parameters, the tool will replace either the command or any one of the parameters with a random string with a different length. Nevertheless, in case there is no command match, there will not be parameter match. So, the target line will be substituted, or a random string will be added to it with a random length.

For example, if a line contains a command and two parameters and the command matches one of the commands in the grammar, the tool will analyze each parameter and change any number in it to "[0-9]". Those parameters are then compared with the parameters in the grammars. If there is a match, then the tool considers the parameters are numeric. After that, they are changed to random numbers with different length. For example, in Listing 1, after the line "perms README.txt 2" is split, the tool will get ['perms','README.txt','2']. The tool will go to first and second parameters. If there is any number in them, the tool will change it to "[0-9]", so in the end the tool will have the parameters look like ['README.txt','[0-9]'], and then it will have a match with a command in Listing 3.4 . Thus, the tool will be able to change the "[0-9]" with a random

number when generating fuzzing files. On the other hand, if there is no numeric parameter and no match, they will be considered as string parameters and will be substituted with a random string. For example, if there is a two-parameter command on a line, the tool can make one of the following replacements:

- Replace the command with a random string and keep the two parameters with no change.
- Keep the command and second parameter with no change and replace the first parameter with a random string.
- Keep the command and first parameter with no change and replace the second parameter with random string.

Since the grammar of the input was extracted, the tool is able to use it to generate more effective fuzzing files by substituting only part of a line. For example, the line *move r3, r2* may be changed to *move r123485792047563808 01, r2* or *move r3, r385972939754098840242* or even *move r3, r-328495723975892*. Helped by the grammar, the string can be substituted precisely. In addition, it is more likely to trigger a bug in a program compared to substituting the whole line with a random string which may not meet the format requirement and hence will be rejected by the program.

4.2 Generating Random String

After finding a target line and matching with the grammars, the tool is going to generate a random string or random number to the selected element from the target line. To have a better coverage of different kinds of random strings, the strings are not randomly generated. Instead, we feed the random test numbers to the random string

generator to specify the format of random string we want. The random test number tells the length, the character types of the random string, and how to replace the target line.

4.2.1 Random Test Numbers

In the beginning, the tool uses random test numbers to specify the length, string type of the random string, and the location in the line to replace the random string. The random test number is three-digit number.

Random test number list is created for each command that is found in the grammar. Each random test number is a three-digit number between 000 and 999. Each of the digits is used for different things in generating a random string. The first digit is for selecting an element that is going to be replaced with a random string; the second digit is for the type of string that is going to be generated; and the third digit is for the length of the generated string. For example, if a random test number is 234, 4 is the first digit, which is for selecting an element to be replaced; 3 is the second digit, which is the type of string to be generated; and 2 is the third digit, which is for the length of generated random string. The goal of the random test number list is to get a better coverage of string types, substitution types, and lengths so the random test number will have the information of string types, substitution types, and lengths, which will be passed to the random string generator to generate different combinations of strings. Moreover, these numbers can be positive or negative, with the negative numbers representing the generation of negative numbers for only numeric parameters. For example, if there is a two-parameter command type *"copy dowjdh file.txt"*, the tool will have a list of 100 different random test numbers generated for command *"copy"*:

copy:[411,-300,-325,32,....]

411 from the list is picked. From the random test number 411, the tool knows first digit 1

is for replacing the first parameter as shown below. The second digit is 1 which is for selecting the string type, which is characters only (upper and lower cases), to be generated. The third digit 4 is for the length of the generated string which is 32.

copy **dfgAjhDGjhjgGDHKgKDGKDgiuioFGHop** file.txt

32 from list is picked. 32 from the list is picked. From the random test number 32, the tool knows first digit 2 is for replacing the second parameter as shown below. The second digit is 3 which is for selecting the string type, which is characters and numbers, to be generated. The third digit 0 is for the length of the generated string which is 4.

copy dowjdh **y2g9**

-300 from the list is picked. From the random test number -300, the tool knows first digit 0 is for replacing the command as shown below. The second digit is 0 which is for selecting the string type, which is numbers only and because the random test number is negative, the tool will generate negative random number. The third digit 3 is for the length of the generated string which is 4.

-34657219 dowjdh file.txt

After completing the generating process from one of the picked random test numbers for a command, the number will be removed from the list. Then, the generating process will continue for that command until all of the random test number list is empty. If the random test number list for a command is empty, one more random test number list will be created for that command to keep generating new random strings for it (if the command is found in coming lines until the last line in the last sample input file). Therefore, the tool will have large combination of different random strings of different length that will be generated for fuzzing files, so that the fuzzer will generate adequate amount of fuzzing files to get a higher probability of finding bugs or vulnerability in a

program.

4.2.2 Digit 1: Selecting and Replacing Part of A Line

For selecting and replacing a string, a random test numbers list for a command is obtained. Then, when there is a command match with a command from the target line, a random test number from the list is picked. As mentioned previously, the first digit is for selecting the element that is going to be replaced with a random string. The choices are going to be 5 options based on the line format (command only or command with parameter(s)) ranging from 0 to 4. These options are:

- 0 means replace the command.
- 1 means replace the first parameter.
- 2 means replace the second parameter.
- 3 means replace the third parameter.
- 4 means select one of the four locations and insert a random string in the selected position.

4.2.3 Digit 2: String Types To Be Generated

In the picked random test number, the second digit specifies the type of string that is going to be generated. The following are the types of strings that are going to be generated by the fuzzer:

- Numbers only.
- Characters only (upper and lower case).
- Symbols only.

- Numbers and characters.
- Numbers, characters, and symbols.
- 0 with space.
- 1 with space.
- %x.
- %n.
- %p.
- Multiple spaces “\x20”.
- Multiple “\xff”.
- Multiple of null character “\x00”.
- Hex control characters.
- Hex characters without the control characters.

The string type 1 with space and 0 with space are for programs that trigger an overflow vulnerability in them. The string types %x, %n, and %p are for discovering format string vulnerability. The hex characters “\x20”, ”\x00”, and “\xff” are going to be used to fuzz the programs for memory corruption. In addition, the hexadecimal control characters from “\x00” to “\x20” and “\x7f” are bad characters that can cause memory corruption in programs under test by giving mixed strings from these characters and then inserting them into a program to trigger a buffer overflow or a bug that crashes the program. Finally, the hexadecimal uncontrol characters from “\x21” to “\x7e” will be used to fuzz

the programs with different combinations of characters to increase the chance of triggering a buffer overflow vulnerability in the fuzzed program.

The tool is able to generate any combination of strings for every generated line because it uses the random test numbers for each command in the fuzzer which will help the tool to create different random string in each round.

4.2.4 Digit 3: Length of A Random String

After the tool has the type of replacement and the type of generated string, it has to determine the length of the generated string. Therefore, the third digit in the picked random test number is for the length of generated string. Because the random test number is three digits from 000 to 999, the third digit in the random test number is in range 0 to 9. The tool will have different kinds of lengths, and there are two lists each containing 10 different lengths. The first list is [4,6,8,16,32,64,128,256,512,1025] and the second list is [1200,1500,1800,2000,2200,2500,2800,3000,3500,4001]. The tool is going to try different lengths to find a bug or vulnerability in the program. The tool works by randomly choosing which list to use. The purpose of having various lengths is to have a higher chance to trigger buffer overflow vulnerabilities. Many programmers forget about checking length or size of a string or an array that could be revealed by fuzzing, so the tool will help find these mistakes during the testing process.

4.3 Fuzzing Process Explanation With An Example

For more clarification, as an example, consider the picked number is 463 and the line is a command with three parameters. So, the first digit 3 means replace the third parameter with a random string. Second digit is 6, which is for the type of the generated string, and it is 1 with space. Moreover, the third digit is 4, which is going to determine the length of the generated string. If list 1 is selected, the length of the random string will be 32;

otherwise, the length of the random string will be 2200.

Once the random string is generated, it will be plugged into the new line, and the new fuzzing file will be constructed. This step will be performed many times to generate a large number of fuzzing files. After completing and generating fuzzing files, the tool will have thousands of fuzzing files, and it will use them to test the target program. Moreover, it will obtain different kinds of fuzzing data in each generated fuzzing file because the random data covers different combination of numbers, letters, and symbols, and for each combination, there will be different string lengths. In the end, there should be a large number of generated fuzzing files, and they can be fed to the target program for testing.

Experimental results for this version of the fuzzer are shown in Section 6.4. However, before we show those results, we introduce an enhanced version of this fuzzur in Chapter 5.

Chapter 5: Markov Chains

In the previous chapter, we proposed a fuzzer which can analyze the format and structure of sample input files and obtain the grammars from them automatically. Then, guided by the extracted grammar, sample input files are modified one line at a time to generate the fuzzing input files. The advantage of the developed fuzzer is that it can extract grammars and use them to guide the fuzzing process quickly and effectively. However, the biggest limitation of the fuzzer, similar to many existing fuzzers, is that the tool is limited to the quality of the sample input files. For example, if there is only one or two sample input files provided, it limits the quality of the extracted grammars. In addition, by making modifications on the sample input files, the fuzzer sticks with the order of commands in the given sample input files, which makes it greatly limited by the contents of the provided sample input files. This situation causes the tool to frequently visit the same locations or go through the same paths in the code, which hinders the tool from exploring paths or locations in the code that could find a vulnerability.

Learning the order of the commands and generating new fuzzing input files without modifying the sample input files are the goals of our study in this chapter. To achieve that, we used Markov chain model on top of our previous tool to help generate new fuzzing input files. By learning the order and probability of the commands from sample input files, the new fuzzer can generate fuzzing input files based on that information and the extracted grammar. Not limited by the order of commands in the sample input files, the new fuzzer goes deeper in the program and thus may reveal more bugs and vulnerabilities.

5.1 Background

Since the idea of Markov chains is based on probabilities, it is an arbitrary process for potential cases that focuses on transition from one state to another based on a probability of each state [60].

The random state variables can be $\{X_0, X_1, X_2, ...\}$ where X_t is state space at time t. Therefore, X_{t+1} depends only on X_t, which means the future state depends on the current state. The state space of Markov chain is all of the states in the process such as $S = \{X_0, X_1, X_2, X_3\}$. The sequence of the process of Markov chain is a particular set of values for $X_0, X_1, X_2, ..., X_n$. For example, if $X_0 = 4$, $X_1 = 7$, $X_2 = 8$,..., $X_n = N$ then the sequence up to time $t = 2$ is 4, 7, 8. So, the sequence is the path from one state to another until last state.

Then, a transition matrix P_t for Markov chains $\{X\}$ at time t is a matrix hold probabilities information of transitioning from current state to next state [60]. It is the most important step for analyzing the Markov chain.

$$X_t \begin{Bmatrix} list \\ all \\ states \end{Bmatrix} \updownarrow \overbrace{\begin{pmatrix} \leftarrow \text{all states} \rightarrow \\ insert \\ Probabilities \\ p_{ij} \end{pmatrix}}^{X_{t+1}} \text{ each row adds to 1}$$

p_{ij} is the transition matrix and $p_{ij} = P$. In the transition matrix, rows indicate current state (X_t) and columns indicate future state (X_{t+1}). The entry (ij) is a conditional probability that means future state is j given that current state i, which is the probability of transitioning from state i to the the state j. Specifically, by providing an ordering of a matrix's rows and columns by the state space S, the $(i, j)^{th}$ elements of the matrix P_t is given by

$$(P_t)_{i,j} = P(X_{t+1} = j | X_t = i)$$

Which means the probability of next state depends on the current state. So, the entries must not be negative and each row i in the matrix must sum to 1. P must have all possible states in the space state S. The P matrix is a square matrix $P(N \times N)$.

5.2 Related Work

There are several studies that employ Markov chain model in generating test cases. Some of them are used to create test cases for testing programs to discover defects, bugs, or vulnerabilities. This section summarizes a representative set of these studies.

Cao et al. [61] introduced an approach to generate test cases automatically based on Markov chain model with reward process that newly consists of a reward method for test results to guide the generation of test cases. The authors use N-step algorithm in this approach combined with Markov chain model, which supports the generation of test flow with the highest probability of triggering software defects or bugs as fast as possible. A HTTP requests for e-commerce system has been collected. Then, Cao et al [61] used them to generate new test cases to test the system. With using Markov chain model, a large number of test cases are generated. They concluded that the average of the found defects is 82.63% which is higher than the manual test which is 65.77%.

Böhme et al. [62] introduced a gray box fuzzing tool AFLFAST that is coverage based and utilizes Markov chain on top of AFL fuzzing tool. It generates test cases without any program analysis. It uses Markov chain model to produce test inputs to test programs. The tool tries to convert generating test cases that exercise high frequency paths to low frequency paths. Markov chain model works by specifying the probability of generating fuzzed test cases that mostly exercise path i to exercise path j, which has low execution or is not executed. They found that AFLFAST is on average 19 times faster than AFL to trigger vulnerabilities. Experiments with three targets show that AFLFAST found 9 vulnerabilities that were assigned CVEs and 3 bugs.

Ouyang et al. [63] presented a fuzzing technique based on Markov chain. It optimizes the test case samples to minimize the generated test cases. It uses instrumentation

to keep track of the code execution information. After that, it employs Markov chain for transitioning from a path to another in the program according to the probabilities. Eventually, Markov chain is used to find the change of the execution path and help a tester to select better test sample case for mutation. After that, the tester can generate test cases and start testing the target program. Experiments showed that the proposed method can support fuzzer to generate effective test samples. Ouyang et al. [63] stated that they found 51 vulnerabilities in application such WPS, with an increased code coverage about 49% in comparison with zzuf. Moreover, the average of exception discovery rate had increased to approximately 9 times in contrast with MiniFuzz.

Wang et al. [64] introduced a new technique for software reliability test case design which is based on using Markov chain model. Authors stated that they used Markov chain model based in UML. Markov chain was explained and created with directed graph that clarifies the process of auto generation of arithmetic of reliability test cases that lead to ease the software reliability test. Wang et al. [64] stated that this method is adequate for generating test cases and efficient for engineering practices.

Zhou [65] presented a method of the generation of Markov usage model for software system and a technique of software reliability test based on it. The author proposed a method to create Markov usage model based on enhanced probability state transition matrix that is table-based framework. Moreover, a software reliability test technique contains generation of test case and based on Markov chain usage model. The author included adequacy determination. In the end, Markov chain test cases generation (MTCG) has been developed which implements the early mentioned methods. Zhou [65] compared between completely random test generation and Markov chain model test generation. In experiments, Markov chain model generated fewer high-efficient test cases than random

test generation which can discover defects on the target application.

Prowell [66] presented a new approach that relies on utilizing concurrency operators to the generated test cases by using simple Markov chain model to create and produce more advanced and complicated test cases to test any software or program. Prowell [66] presents experiments on XML language to show that using Markov chain can have an efficient way raise an event by obtaining a test case for a tester.

Listing 5.1: Commands and Their common Parameters After adding "Rs"

```
{'copy2':[['README.txt','Rs'],['authentication.db','Rs']
,['Rs','Rs']],
'erase1': [['README.txt'],['authentication.db'],['Rs']],
'exit0': [],
'first2':[['README.txt','10'],
['authentication.db','10'],['Rs','Rs']],
'last2': [['README.txt','10'],
['authentication.db','10'],['Rs','Rs']],
'list0':[],
'logout0': [],
'make1':['Rs'],
'perms2':[['README.txt','Rs'],['authentication.db','Rs'],
['README.txt','[0-9]'], ['authentication.db','[0-9]'],
['Rs','Rs']],
'show1':[['README.txt'],['authentication.db'],['Rs']],
'write1':[['README.txt'],['authentication.db'],['Rs']]}
```

5.3 Adding Random String Indicator "Rs" to Grammars

Before applying Markov chain, the grammars need to be updated and added "Rs" that indicates "Random string". Therefore, the purpose of this step is to update the grammars in Listing 3.4 to add "Rs". This helps to ensure the tool can have each command that has missing parameters to get complete parameters number for that command; therefore, supporting to newly generate complete commands with parameters as shown in Section 5.4.2

After obtaining the grammars in chapter 3, the final grammar will be further updated

to add "Rs" for the commands that have no or less fixed parameters. For example, in Listing 3.4, the ***make1*** command (on line 8) has no parameter found after it. It shows that it is a one-parameter command. Therefore, "Rs" is added to it. Similarly, ***perms2*** and ***copy2*** are two-parameter commands which should have two parameters after them. However, from the extracted grammar, only the first parameter is fixed to "README.txt" or "authentication.db", and the second parameter is probably a random string. To represent random string, "Rs" is added as the second parameters for ***perms2*** and ***copy2***. As shown in Listing 5.1, extra "Rs" parameters are added for each command based on the type of command. What is more, it helps calculate the remaining random string parameters for each command in later steps to support the generation of new fuzzing input files.

5.4 Using Markov Chain Model in the Fuzzer

The Markov chain model is used on top of the previous tool, where we assume the fuzzer has analyzed and extracted the grammar from sample input files. By applying Markov chain model, the tool further analyzes and studies the sample input files to collect information about the orders and probabilities of each command. There are three major stages: (1) Analyzing commands and parameters by learning the command order by using Markov chain model, (2) Calculating command and parameter probabilities, and (3) Generating completely new fuzzing input files with all of the information learned from analyzing sample input files.

5.4.1 Analyzing Commands and Parameters Using Markov Chain Model

In this stage, the tool obtains commands from sample input files and applies Markov chain model to get the possible commands order from analyzing all of the sample input files.

5.4.1.1 Extracting Unsorted Commands

This step extracts commands from sample input files without sorting so the tool can keep the order of commands unaffected. The tool starts getting the first words in each line and saves them based on the order of occurrence in sample input files. For example, if a sample input file has the commands "copy, list, print, read, write, exit" and another sample input file has "read, write, copy, list, print, exit", that tool will have two lists created: [copy, list, print, read, write, exit] and [read, write, copy, list, print, exit]. In addition, to know the first and last command in each sample input file, a start mark or pattern "START" and an end mark or pattern "FINAL" are added to each command lists. If the tool does not know the start and the end of the commands list, it will get unordered commands and will not stop producing the commands, as explained later in section 5.4.1.3. In the previous example, the two commands lists are changed to [START, copy, list, print, read, write, exit, FINAL], and [START, read, write, copy, list, print, exit, FINAL].

5.4.1.2 Calculating Commands Transition Probabilities

In this stage, the tool establishes a transition matrix for the extracted commands and calculates the probabilities of each next command (*transition to*) for each current (*transition from*) command. The tool takes each current command and next command and saves them in a transition pair, *(current, next)*. In this way, a list of current and next commands transition pair looks similar to (command_1, command_2), (command_2, command_3)...., (command_n, command_{n+1}). These pairs are used to create transition matrix for current commands and next commands. After that, the tool calculates transition probability of each next command for each current command.

The tool first counts the number of *transition from* commands (current commands) as

well as the occurrences of *transition to* commands (next commands). Then it counts the number of occurrences for each transition pair. With these numbers, the tool calculates the probability of each next command by dividing the occurrences of each next command by the occurrence number of the current command. The transition probabilities of all commands will be stored and kept for further processes. For example, if the commands list is *[START, list, copy, list, print, list, print, list, exit, FINAL]*, a list of transition pairs looks like *(START, list), (list, copy), (copy, list), (list, print), (print, list), (list, print), (print, list), (list, exit), (exit, FINAL)*. From the list, we can tell that command *list* is followed by commands *copy*, *print*, *exit* at probabilities of 25%, 50%, and 25% correspondingly. Command *copy* is always followed by command *list*; similarly, command *print* has command *list* as the next command at 100% probability.

The Markov chain model in our fuzzer keeps probability of each next command for a current state command to be used when generating new inputs. So, the current state command has one or more commands following it with their probabilities. When a command is found, the tool chooses the next command based on the probability of that command. The tool repeats these steps for upcoming commands until it stops.

5.4.1.3 Obtaining Next Command

After the tool learns the probabilities of next commands for a particular current command, it can select the next command for a specific current command based on the transition probabilities. To do this, starting from "START", which is considered the current state i, the tool chooses the next command j based on its probability. The current command maintains a pool of next commands and their probabilities of picking the next command. Once the next command is chosen, it becomes the current command, and similarly, it has a group of possible next commands and their probabilities. Then

the tool keeps selecting commands based on their probabilities until it reaches pattern "FINAL" and the command generation stops. When "FINAL" is reached, the tool saves the generated order of commands as a possible order of commands which is called commands chain. A chain of generated order of commands may look like [START, $command_1$, $command_2$ ..., $command_n$, FINAL]. Since "START" and "FINAL" are for helping the tool to know the start and the end commands in each sample input file, they are discarded from the command chain.

For each list of command chains, the tool does not have a fixed number of commands because the generation of a list only stops when reaching the pattern "FINAL". Picking the next command is all based on the probability. Therefore, for the same program, sometimes it may take only five commands to reach to "FINAL", but sometimes it may select more than 100 commands before reaching "FINAL".

At the end of this stage, the tool generates many possible lists of command chain. These lists of command chain will be used for generating new fuzzing files in the later stage.

5.4.1.4 Analyzing and Counting Parameters

After analyzing the commands, the fuzzer starts analyzing parameters. The goal of this stage is to find out how many parameter matches (from sample input files) there are for each command in the grammar and the probabilities of parameters. From the results of the parameter analysis, the tool will understand how to pick parameters for each command.

The tool accomplishes this by first counting all parameter matches for each command in the grammar which match a command in the command chains. For each command, no matter if it is a one-parameter, two-parameter, or three-parameter command, the tool

takes all parameters (one, two, three parameters) for that command from sample input files. Then, each parameter from sample input files is compared with the parameters in the grammar; if there is a match, then the fuzzer counts and collects it for future calculations.

Then, the fuzzer begins checking whether or not there is a "[0-9]" parameter in the grammar parameters for each command. If there is a "[0-9]" parameter, it analyzes the parameters from sample input files and changes any number to "[0-9]" and counts the matches with the grammar. If there is no "[0-9]" in the grammar parameter, the tool takes the parameters, finds any match for parameters in the grammar, and counts them. Moreover, if there is "Rs" in the parameter grammar, the tool changes the strings at that location to "Rs" in the sample input files. It counts any match between the parameters in the grammar and the parameters in the sample input files. Then, the tool subtracts the command occurrence number from the parameter grammar total count number for parameters contain "[0-9]", "Rs", and other parameter grammars that has been calculated previously. The result of that calculation will be for the "Rs" parameters if the "Rs" parameters is one of the following command types: one-parameter type ["Rs"], two-parameter type ["Rs","Rs"] or three-parameter type ["Rs","Rs","Rs"].

While scanning and analyzing parameters, the tool also collects the total number of all parameters for each command in the grammars.

5.4.1.5 Calculating Grammar Parameters Probabilities

After the tool gets the total count numbers of each parameter for each command in the grammar based on the command chains, it calculates the occurrences of each command in the sample input files. Then, it divides the parameters' total count numbers by the calculated number to get each parameter's probability. By having the parameter's

probability data, the tool knows how to choose a parameter for a command based on the probability of parameters.

At the end of this step, each parameter has a probability associated with it. This information will support the process of picking parameters for each command.

5.4.2 Generating New Input Lines

The generation of new input lines consists of two steps. First, the tool employs commands chain and parameters probabilities to generate new input lines. Second, it substitutes the numeric and string parameters with random number and random string. Then, the tool uses these new input lines and generates completely new fuzzing input files.

5.4.2.1 Obtaining New Input Lines by Using Chain of Commands and Parameters Probabilities

The goal of this stage is to generate new input lines based on the grammars and probabilities with the use of possible chain of commands (generated in Section 5.4.1.3). The chains of order of commands are an important component in this stage because they indicate commands orders.

Taking each chain of commands, the tool picks parameters for each command in the chain. If the command in the chain matches a command in the grammar, the fuzzer chooses the parameter for that command based on the probability. Then it appends the selected parameter to the command and saves it as an input line. The tool repeats this for upcoming commands in the chain.

For any command in the chain list that is not in the grammar, the tool adds "Rs" parameters to indicate a random string is needed. If the command is not in the command chain list, the tool fills the remaining line with certain number of "Rs" parameters ("Rs" means random string). For example, command "Print5" and "Read2" tell that

the command “Print” has five parameters following it and “Read” has two parameters following it. If command “Print5” and “Read2” are in the list of command chain list but do not have any information on the parameters probabilities for those commands, command “Print5” will have [“Rs”,“Rs”,“Rs”,“Rs”,“Rs”] appended to the command and “Read2” will have [“Rs”,“Rs”] appended to the command, too.

Finally, the fuzzer saves the input lines in a file which will be used to generate input files. These input format files tell the fuzzer how to generate an input file. At the end of this stage, a large number of input format files will be ready for later processing.

5.4.3 Generating Completely New Fuzzing Input Files

After getting the input format files from previous steps, the tool will use them to generate new fuzzing files. The overall method of generating fuzzing files is identical to the techniques and methods as discussed in our former tool (see Chapter 4). Briefly, it takes one of the input format files and changes only one line and keeps other lines unchanged. This stage is going to be repeated many times until the tool generates about 5000 completely new fuzzing input files.

Before substituting a line, the “[0-9]” and “Rs” in the format will be replaced with a random number or a random string respectively by using random test number. The random test number is different than the random test number in Chapter 4. The different is that the test number used here is 2-digit. First digit is for string type and the second digit is for selecting a length from this list [4, 6, 8, 16, 32, 64, 70, 80, 90, 100]. So, If it encounters a “[0-9]” on a new input line, a random number is generated and replaces “[0-9]”. Similarly, when finding a “Rs”, it will be substituted with a random string.

The tool starts with the first line and analyzes the line. It compares the command on that line with the commands in grammar. If there is a match, it compares the parameters

to find out whether there is a match or not. If there is a match, the tool will substitute the command or the parameters with a random string or a random number. In next round, it will target the next line in the same input format file and continue until it reaches the last line in this file. Then it move to the next input format file and starts substituting from the first line again.

When generating a random number or random string, the method of random test number is used. The random test number tells the random number generator what kind of random string or number is expected, such as the type of the string and its length. By using a random test number, the tool can generate different kinds of random strings.

If the sample input file has hexadecimal characters more than plain text characters, the tool will create random hexadecimal characters; otherwise, it will create plain text characters in the new input lines that have "Rs". For more details about the random test number see Section 4.2.

The generated input file looks like a real input file. This new input file can be used to generate new fuzzing files by using the fuzzing technique mentioned in Chapter 4.

Chapter 6: Experiments and Evaluation

This chapter discusses the experiments that have been conducted to evaluates the fuzzer and presents the findings and learned lessons. To test each fuzzing input file generated by the fuzzer, a bash script runs to go through all of the programs in the dataset and feed in fuzzing input files to each program. Then, it looks for crashed program by scanning for core dump or segmentation fault error messages. After that, it collects the findings and results. The 24-hour test was one of the evaluations conducted. Then, there was a comparison between the tool and others. Moreover, the tool was tested with respect to the code coverage of each program. In addition, percentage numbers were adjusted to evaluate the tool with the best percentage that have better code coverage and number of crashes.

6.1 Experiments Setup and Running

6.1.1 Segmentation Fault (SIGSEGV)

Internal operating system, Figure 6.1 shows that a compiled C program's memory consists of five major segments. Text segment for code segment that includes machine language instructions of the program. Data segment has the initialized global and static variables [67]. Block Started by Symbol (BSS) segment contains uninitialized global and static variables. Data in BSS is initialized by kernel to arithmetic 0 before executing the program. Heap segment where dynamic memory allocation takes place. This location starts at the end of BSS and grows up to higher addresses. Stack segment is placed under operating system kernel. The stack segment grows toward lower addresses which opposites of the heap segment [67].

Segmentation fault occurs when an address of a variable falls outside the segment that the variable is assigned to. Therefore, after experiencing an error in writing to a memory segment, the Unix or Linux operating system sends a SIGSEGV signal to the program,

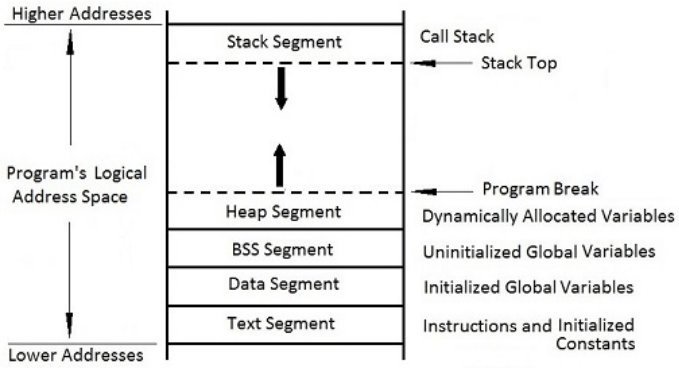


Figure 6.1: Memory Layout.

which then crashes and displays the "Segmentation fault" notification. Segmentation faults are normally specific to high-level languages like C, which ask the programmer to assign memory to an operating code.

As shown Listing 6.1, integer ***n*** is an integer input by a user. If it is more than 1000, it is going to result in segmentation fault because it accesses memory space beyond the array boundary. Without checking the user input, the program may lead to illegal memory access and cause the program to crash.

Listing 6.1: Illegal Memory Access1

```
1 int n;
2 int array[1000];
3 printf("Enter an integer number: ");
4 scanf("%d",&n);
5 for (int i = 0; i < n ; i++)
6   array[i] = i;
```

Listing 6.2 is another example of illegal memory access. No memory space was allocated and pointed by pointer ***b***, so ***b*** will point to a random location in the memory. Therefore, accessing ***b[0]*** may cause segmentation fault.

Listing 6.2: Illegal Memory Access2

```
1 float *a, *b;
2 a = (float*)malloc(1000);
3 b[0] = 1.0;
```

6.1.2 Experiment Setup

The research experiments begin with analyzing sample input files from DARPA CGC programs to get grammars from them because the tool needs to get more knowledge on the format of the sample input files. After that, the tool generates fuzzing input files to be used in testing process. When testing the fuzzing input files, we look for segmentation fault or program crash.

```
hamadadmin@cs-rasp19js04:/datadrive/cb-multios/build/challenges/Printer$ ./testscript.sh
TESTING: testcases1/127.0.0.1.10364-127.12.132.108.2096_37152.pov
CORED: testcases1/127.0.0.1.10830-127.12.132.108.2096_66099.pov
TESTING: testcases1/127.0.0.1.10841-127.12.132.108.2096_98722.pov
TESTING: testcases1/127.0.0.1.11075-127.12.132.108.2096_23759.pov
TESTING: testcases1/127.0.0.1.11579-127.12.132.108.2096_78859.pov
CORED: testcases1/127.0.0.1.11579-127.12.132.108.2096_78890.pov
TESTING: testcases1/127.0.0.1.12919-127.12.132.108.2096_22616.pov
TESTING: testcases1/127.0.0.1.13246-127.12.132.108.2096_71973.pov
```

Figure 6.2: Printer Program Core Dumped Example.

A bash script has been written to loop through each fuzzing file and feed them to the target program under test. In the meantime, the output of running the program is monitored for any "core dumped" or "Segmentation fault".

```
hamadadmin@cs-rasp19js04:/datadrive/cb-multios/build/challenges/payroll$ ./testscript.sh
CORED: testcases1/127.0.0.1.10080-127.241.131.136.2813_334100.pov
TESTING: testcases1/127.0.0.1.10080-127.241.131.136.2813_334100.pov
CORED: testcases1/127.0.0.1.10080-127.241.131.136.2813_334101.pov
CORED: testcases1/127.0.0.1.10080-127.241.131.136.2813_334102.pov
TESTING: testcases1/127.0.0.1.10080-127.241.131.136.2813_334102.pov
CORED: testcases1/127.0.0.1.10080-127.241.131.136.2813_334103.pov
TESTING: testcases1/127.0.0.1.10080-127.241.131.136.2813_334104.pov
TESTING: testcases1/127.0.0.1.10080-127.241.131.136.2813_334106.pov
CORED: testcases1/127.0.0.1.10080-127.241.131.136.2813_334108.pov
TESTING: testcases1/127.0.0.1.10080-127.241.131.136.2813_334108.pov
CORED: testcases1/127.0.0.1.10080-127.241.131.136.2813_334109.pov
TESTING: testcases1/127.0.0.1.10080-127.241.131.136.2813_33410.pov
```

Figure 6.3: Payroll Program Core Dumped Example.

When a message "core dumped" or "Segmentation fault" is displayed in the output, it means a crash in the program has happened; then the running program stops and terminates the running process. "Segmentation fault" message informs that the program encounters a serious error and the program has a bug or vulnerability. As shown in Figures 6.2 and 6.3, when a fuzzing input file crashes a program under test, it is displayed

on the terminal that the program has been "CORED" or crashed, and then it continues testing next fuzzing files.

For instance, as shown in Figure 6.4, when testing the RRPN program with the generated fuzzing inputs, the program crashed and displayed that the "Segmentation fault" when the file "127.0.0.144788-127.44.224.53.2457_3780.pov" and "127.0.0.144788-127.44.224.53.2457_3781.pov" were fed to the program. While running other input files, there was no crash or SIGSEGV message.

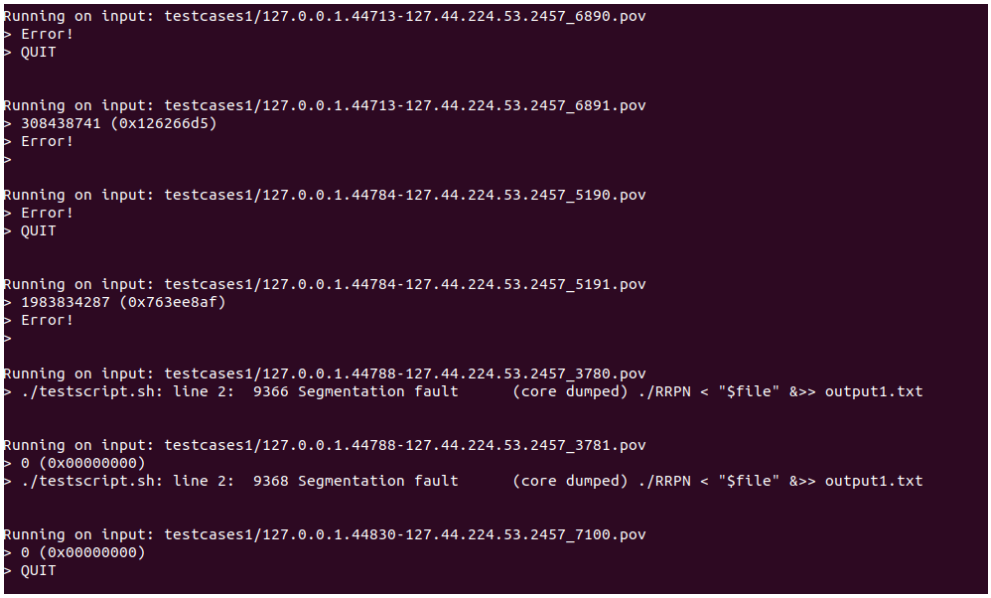


Figure 6.4: RRPN Program Crash.

6.1.3 DARPA CGC Dataset

There are 247 programs in the Cyber Grand Challenge dataset. We use CB-multios dataset that was ported and provided by TrailofBits team [68]. We excluded 12 programs because of some issues in compilation and running. The following programs have compilation issues:

- CAT
- CROMU_00038

- Ghost_In_The_CGC
- SBTP
- Trust_Plateform_Module
- adventure_game

Also, some programs had programming and running issues. Some of them crash in the beginning of running and the others crash after inserting an input. Part of the problems we saw was things like uninitialized variables. The DEGREE environment initialized the stack to all zeros, but normal Linux does not. This resulted in some errors when porting. The following programs have execution problems:

- A_Game_of_Chance
- LAN_Simulator
- Messaging
- Mount_Filemore
- middleware_handshake
- QUIETSQUARE

6.2 Experiments Overview

Different kinds of experiments has been conducted to evaluate the developed fuzzing tool. The experiments performed on our regular fuzzing tool that has been explained in Chapters 3 and 4. Moreover, another experiment performed with the developed tool with Markov chain that was explained in Chapter 5.

The following experiments has been conducted in our research:

- 24 hours testing: the testing was conducted on our regular fuzzing tool for 24 hours running to fuzz programs.
- Comparing the tool with other tools: the experiment has been performed to compare our regular fuzzing tool with other fuzzing tools.
- Experiments with code coverage: the experiments have been conducted on our regular fuzzing tool to examine code coverage for each program.
- Adjustment percentages crashes and code coverage: the experiments have conducted on our regular fuzzing tool to examine the best percentage based on number of crashes and code coverage.
- Experiments on fuzzer updated with Markov chain: the experiments have been performed on the updated tool using Markov chain model only.

6.3 24 Hours Testing

The purpose of our 24 hour test was to examine the programs in 24 hours and find crashed programs. The system is Xubuntu 18.04 VM, which has 60 cores, 512 GB of memory, and 80 TB of hard drive space. The testing entails running and testing each program for up to 24 hours. Moreover, the testing of each program was planned to test for up to 24 hours by generating fuzzing files each time when the previous test run out of fuzzing files and continue generating fuzzing files and test them until the timer reaches 24 hours; otherwise, if there is a crash the testing would be stopped. It tested 40 programs in parallel every time. This testing took about 8 days to test all of the programs.

Table 6.1 shows the programs that crashed in this experiment. There are 82 crashed programs that have been found. It consists of five columns: the first column is program name, the second column is the generation time to generate fuzzing files and it is in

minutes and seconds, the third column is the number of samples input files, the fourth column is the number of generated fuzzing files, and the fifth is the testing time and it is in day, hours, minutes, and seconds.

For example, in first row, the program name is 3D_Image_Toolkit. Its generation time is 17 seconds, the number of sample files is 1000 files, the number of generated fuzzing files is 6915 files, and the testing time is 10 hours, 37 minutes, and 43 seconds.

Table 6.1: List of 24 Hours Run Crashed Programs

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (d:h:m:s)**
3D_Image_Toolkit	0:17	1000	6915	0:10:37:43
Azurad	14:6	1000	196782	0:0:3:47
Bloomy_Sunday	1:16	1000	30900	0:0:0:20
CGC_Planet_Markup_Language_Parser	15:10	1000	300280	0:1:10:24
Charter	0:7	1000	2458	0:0:0:19
Checkmate	0:11	1000	4995	0:0:18:43
CML	2:40	1000	48607	0:0:10:28
Cromulence_All_Service	10:31	1000	225262	0:17:54:34
CTTP	6:1	1000	89278	0:0:27:18
DFARS_Sample_Service	1:43	1000	36657	0:0:0:2
Diary_Parser	0:7	1000	3093	0:0:0:1
Diophantine_Password_Wallet	0:38	1000	19747	0:3:12:15
Divelogger2	3:48	1000	82757	0:0:1:14
Document_Rendering_Engine	9:38	1000	222070	0:0:0:40
Eddy	0:3	1000	1226	0:0:21:18
electronictrading	0:4	1000	1006	0:0:12:36
EternalPass	0:13	1000	6984	0:0:0:0
FablesReport	0:4	1000	1000	0:5:35:4
FileSys	0:31	1000	12929	0:0:2:6
Filesystem_Command_Shell	0:32	1000	36627	0:0:6:3
Finicky_File_Folder	0:31	1000	11366	0:0:0:1
Flash_File_System	0:12	1000	5164	0:0:2:26
Flight_Routes	1:42	1000	46971	0:3:0:52
Fortress	5:1	1000	151818	0:13:20:45

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (d:h:m:s)**
Game_Night	0:28	1000	14618	0:0:0:2
Grit	9:59	1000	147446	0:0:0:10
HackMan	6:30	1000	135426	0:0:0:6
HeartThrob	0:8	1000	2987	0:0:34:22
HighFrequencyTradingAlgo	0:3	1000	1012	0:0:0:30
Hug_Game	0:23	1000	27058	0:0:4:15
INSULATR	4:23	1000	38556	0:3:5:42
Kaprica_Script_Interpreter	19:30	1000	302907	0:0:0:41
KTY_Pretty_Printer	1:33	1000	38319	0:1:19:33
Lazybox	0:58	1000	31622	0:0:0:2
Loud_Square_Instant_Messaging_Protocol_LSIMP	1:54	1000	52134	0:18:39:28
Matchmaker	1:14	1000	35638	0:0:0:3
matrices_for_sale	0:12	1000	6450	0:0:0:12
Monster_Game	1:31	1000	46249	0:17:32:51
Movie_Rental_Service	0:20	1000	9628	0:0:1:41
Multicast_Chat_Server	12:29	968	253273	0:0:0:7
Multipass3	1:9	1000	24299	0:7:38:54
Network_Queuing_Simulator	2:23	1000	52461	0:0:0:1
online_job_application	1:31	1000	34581	0:0:26:30
online_job_application2	1:7	1000	34570	0:3:4:50
OTPSim	2:58	1000	51325	0:0:0:1
Palindrome	0:18	1000	8889	0:0:0:1
Palindrome2	0:20	1000	9731	0:0:0:0
payroll	10:23	900	113370	0:0:0:14
Pipelined	0:16	1000	7600	0:0:1:3
pizza_ordering_system	2:0	1000	46155	0:0:1:14
Printer	2:44	1000	72683	0:0:1:5
PRU	0:11	1000	4810	0:0:0:2
PTaaS	2:55	1000	39967	0:0:0:1
Query_Calculator	0:2	1000	1000	0:0:0:1
Recipe_and_Pantry_Manager	7:57	1000	186476	0:1:2:12
Recipe_Database	9:50	1000	196705	0:0:2:47
REMATCH_2 –Mail_Server–Crackaddr	0:15	1000	4003	0:0:7:11

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (d:h:m:s)**
REMATCH_3–Address_Resolution_Service–SQL_Slammer	0:14	1000	6569	0:0:0:1
REMATCH_4–CGCRPC_Server–MS08-067	0:6	1000	2947	0:21:11:3
REMATCH_5–File_Explorer–LNK_Bug	0:20	1000	9942	0:0:0:23
REMATCH_6–Secure_Server–Heartbleed	0:27	1000	11409	0:0:0:2
RRPN	0:5	1000	1907	0:0:0:11
Sad_Face_Template_Engine_SFTE	0:45	1000	23031	0:0:0:26
Sample_Shipgame	1:48	1000	51385	0:0:0:1
SCUBA_Dive_Logging	3:25	1000	81656	0:0:2:6
Secure_Compression	1:33	1000	46846	0:0:0:1
simple_integer_calculator	0:6	1000	2136	0:0:2:2
simplenote	13:11	1000	300314	0:0:0:10
SPIFFS	1:1	1000	25222	0:0:34:52
stream_vm	22:44	1000	300542	0:0:1:1
TAINTEDLOVE	4:51	1000	69188	0:0:0:21
Tennis_Ball_Motion_Calculator	0:18	1000	9134	0:0:0:43
TextSearch	12:46	1000	242852	0:6:35:14
The_Longest_Road	2:12	1000	48838	0:0:0:10
TVS	31:31	1000	300865	0:0:23:8
university_enrollment	1:0	1000	27849	0:0:3:2
Vector_Graphics_2	0:11	1000	2377	0:0:4:46
Vector_Graphics_Format	0:5	1000	2371	0:0:0:5
virtual_pet	0:9	1000	4445	0:0:0:13
Water_Treatment_Facility_Simulator	2:53	1000	59185	0:1:39:16
WhackJack	2:47	1000	55939	0:0:0:8
WordCompletion	0:8	1000	3926	0:0:0:59

6.4 Comparing Our Tool with Other Tools

6.4.1 Used Tools in Experiment

We compared our work with these four tools: the latest version of AFL [13], MOPT [69], FairFuzz [70], and AFLFAST [62].

AFL (American fuzzy lop) is a state-of-the-art greybox fuzzer [13]. It uses trivial instrumentation to get new path coverage information. Based on that, AFL can choose unique identification of the path that is applied by an input. Then, it utilizes genetic algorithms to find interesting test cases that are likely to reveal new internal states in the program under test. After that, these test cases will be added to the sample inputs queue.

Lyu et al. [69] introduced MOPT a mutation-based fuzzing tool. MOPT uses a customized Particle Swarm Optimization (PSO) algorithm to explore the possibility to give an optimal selection probability of different kinds of mutation operations. The optimization enhances the ability of a fuzzer to find the coverage information quickly.

Lemieux et al. [70] stated that FairFuzz is a mutation-based gray-box fuzzing tool. The tool first looks for these branches that are rarely hit by fewer AFL inputs. Second, based on new mutation operation techniques, it makes the tool lean to generate inputs hitting a provided rare branch. This mutation is calculated dynamically during fuzzing and can be used to fuzz other targets.

According to Böhme et al. [62], AFLFAST is a graybox fuzzing tool that uses Markov chain knowledge [60]. It does not require a program analysis. It generates new test inputs with few mutations of seed input samples. It employs Markov chain model that specifies the probability of fuzzing the sample input that exercises path i, which then provides an input that exercises path j. Instead of fuzzing highly visited locations, the tool redirects to fuzz lower visited locations in the code.

6.4.2 Experiments Setup

The machine for generating fuzzing files and testing has environment: the CPU is AMD Ryzen Threadripper 2920X 12-Core Processor, memory is 32 GB, Ubuntu 18.04.4 LTS, and OS type 64-bit.

We use CB-multios dataset that is provided by TrailofBits team [68] who ported DARPA dataset from DEGREE system to Linux. There are about 247 programs. We excluded 12 challenges because there is an issue with the compilation and running of those programs. Around 235 programs had been tested separately with different number of rounds for a one-hour run. During the one hour, the testing will be stopped if a crash is found.

AFL [13], MOPT [69], FairFuzz [70], and AFLFAST [62] fuzzing tools had been tested with 235 programs. Each one took 1 hour of testing for each program. After tests were completed, the number of crashed programs was collected.

6.4.3 Testing Results and Observations

As a result, our fuzzing tool discovered vulnerabilities more than other tools. AFL crashed 45 programs, which is 19.14%. AFLFAST crashed 44 programs, which is 18.72%. Moreover, MOPT and FairFuzz crashed 54 programs, which is 22.97%. Our tool crashed 79 programs, which is 33.61% of the programs. With the help of grammar, our tool found 34 more than AFL, 35 more than AFLFast, and 25 more than MOPT and FairFuzz.

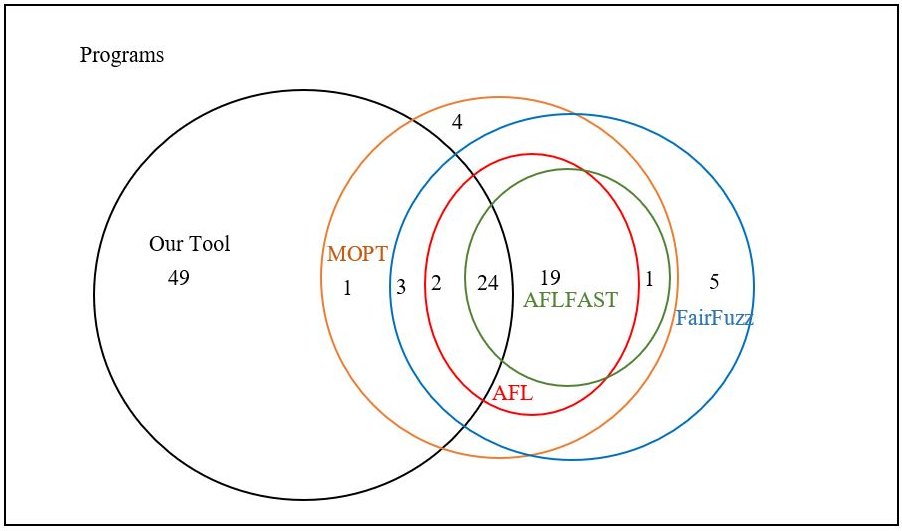


Figure 6.5: Venn Diagram for Discovered Bugs

As the rsualt in Figure 6.5, it shows that our tool outperforms the other tools. So, in comparison with AFL, our tool crashed 26 same crashed programs that AFL had crashed and 53 programs did not crash. Also, AFLFAST has 24 same crashed programs with our tool but AFLFAST did not crash the 55 programs that our tool crashed. MOPT tool has 30 same crashes as our tool crashes; however, It did not find the 49 that our tool crashed. In case of FairFuzz, our tool and FairFuzz crashed 29 same programs but it did not crash the 50 programs that our tool crashed. Moreover, the fuzzing techniques have different kinds of options to create fuzzed inputs that support revealing bugs and vulnerabilities. For example, it can use the grammar for numerical parameters and replace it with a random number. Also, the grammar for string parameters supports obtaining different types of strings with different lengths that can provide the fuzzing tool with a higher possibility to trigger buffer overflow, off-by-one error, use after free vulnerabilities, etc.

We observed that our tool crashed programs faster than others. For most of the crashed programs, we noticed that AFL took 20 to 30 minutes to crash a program. Moreover, AFLFast found most vulnerabilities in 10 to 15 minutes. FairFuzz discovered

vulnerabilities in most of programs in 25 to 30 minutes. Also, MOPT needs an average of 15 to 20 minutes to crash a program. However, our tool usually finds a crash in 5 minutes.

Table 6.2 shows the crashed programs. The table has five columns. The first one is the program names. The second column is generation time, which is the duration time for generating all of the fuzzing files. It is based on minutes and seconds (m:n). The third column shows the number of sample input files for each program. The fourth column shows the number of generated fuzzing files, and the fifth column is testing time, which is the duration of time the program has been tested by the generated fuzzing files until a vulnerability has been found by the tool, and it is based on minutes and seconds (m:s).

Table 6.2: Crashed Programs by our Fuzzer

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (m:s)**
Azurad	3:44	1000	196782	1:45
Bloomy_Sunday	0:26	1000	30900	1:4
CGC_Planet_Markup _Language_Parser	4:43	1000	300018	42:8
Charter	0:2	1000	2458	4:43
Checkmate	0:5	1000	4995	18:46
CML	0:50	1000	48607	2:32
Cromulence_All_Service	6:43	1000	225262	34:6
CTTP	5:41	1000	89278	59:59
DFARS_Sample_Service	2:32	1000	36657	0:41
Diary_Parser	0:3	1000	3093	0:50
Diophantine_Password _Wallet	0:16	1000	19747	48:42
Divelogger2	1:28	1000	82757	5:53
Document_Rendering _Engine	3:37	1000	222070	0:30
Eddy	0:1	1000	1226	9:2
electronictrading	0:1	1000	1006	21:53
EternalPass	0:5	1000	6984	2:1
FablesReport	0:2	1000	1000	1:20
FileSys	0:15	1000	12929	0:50

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (m:s)**
Filesystem_Command_Shell	0:0:0:30	1000	36627	0:0:5:12
Finicky_File_Folder	0:14	1000	11366	0:43
Flash_File_System	0:4	1000	5164	1:36
Flight_Routes	0:52	1000	46971	0:53
Fortress	3:2	1000	151818	49:23
Game_Night	0:12	1000	14618	0:56
Grit	3:19	1000	147446	3:10
HackMan	2:16	1000	135426	1:4
HeartThrob	0:3	1000	2987	4:3
HighFrequencyTradingAlgo	0:1	1000	1012	0:57
Hug_Game	0:25	1000	27058	6:29
INSULATR	2:7	1000	38556	34:31
Kaprica_Script_Interpreter	5:37	1000	302474	0:47
KTY_Pretty_Printer	0:33	1000	38319	13:24
Lazybox	0:24	1000	31622	0:35
Loud_Square_Instant _Messaging_Protocol_LSIMP	0:49	1000	52134	24:32
Matchmaker	0:31	1000	35638	0:21
matrices_for_sale	0:6	1000	6450	0:32
Monster_Game	0:40	1000	46249	32:32
Movie_Rental_Service	0:8	1000	9628	0:32
Multicast_Chat_Server	3:59	968	253273	0:34
Network_Queuing_Simulat- or	0:45	1000	52461	0:13
online_job_application	0:29	1000	34581	2:40
online_job_application2	0:29	1000	34570	18:15
OTPSim	0:50	1000	51325	0:12
Palindrome	0:7	1000	8889	0:5
Palindrome2	0:8	1000	9731	0:7
payroll	3:27	900	113370	0:11
Pipelined	0:6	1000	7600	0:20
pizza_ordering_system	0:40	1000	46155	0:35
Printer	1:12	1000	72683	0:42
PRU	0:4	1000	4810	1:1
PTaaS	1:8	1000	39967	0:33
Query_Calculator	0:1	1000	1000	1:9

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (m:s)**
Recipe_and_Pantry _Manager	3:18	1000	186476	9:39
Recipe_Database	3:31	1000	196705	1:53
REMATCH_2–Mail_Server–Crackaddr	0:3	1000	4003	0:25
REMATCH_3–Address_Resolution_Service–SQL_Slammer	0:5	1000	6569	0:8
REMATCH_4–CGCRPC Server-MS08-067	0:2	1000	2947	13:51
REMATCH_5–File_Explorer–LNK_Bug	0:8	1000	9942	0:10
REMATCH_6–Secure_Server–Heartbleed	0:10	1000	11409	0:23
RRPN	0:1	1000	1907	0:29
Sad_Face_Template_Engine _SFTE	0:18	1000	23031	0:9
Sample_Shipgame	0:45	1000	51385	0:36
SCUBA_Dive_Logging	1:14	1000	81656	1:51
Secure_Compression	0:40	1000	46846	0:1
simple_integer_calculator	0:2	1000	2136	1:10
simplenote	4:5	1000	300239	0:5
SPIFFS	0:24	1000	25222	24:41
stream_vm	5:6	1000	300448	0:4
TAINTEDLOVE	1:57	1000	69188	0:11
Tennis_Ball_Motion Calculator	0:8	1000	9134	0:2
The_Longest_Road	0:46	1000	48838	0:14
TVS	5:31	1000	300456	3:24
university_enrollment	0:36	1000	27849	3:44
Vector_Graphics_2	0:3	1000	2377	12:54
Vector_Graphics_Format	0:2	1000	2371	0:23
virtual_pet	0:3	1000	4445	0:12
Water_Treatment_Facility _Simulator	1:11	1000	59185	11:28
WhackJack	1:9	1000	55939	16:11
WordCompletion	0:3	1000	3926	0:55

6.5 Experiments with Code Coverage Feedback

6.5.1 Code Coverage Advantages and Disadvantages

Code coverage is a measurement for a program to find how much of the code was executed based on a standard. Code coverage can help a developer to enhance the testing. Moreover, it can assist with improving the quality of the code. Also, it gives a good view of how much of the code has been tested. However, a good code coverage does not mean good testing [71]. The limitation of code coverage is that it does not guarantee that the 100% code coverage is free of vulnerabilities [71].

There are many types of code coverage. First, line coverage determines how many lines of the code were successfully executed. Second, branch coverage determines how many branches (loop, if-else statements) of the code were successfully executed. Third, function coverage determines how many functions were called. Fourth, path coverage determines that each path from the input to output in the program is executed or covered.

We conducted the code coverage in our experiments to examine the performance of the fuzzing tool in generating fuzzing input files that are inserted in a program and how much of the code has been tested.

6.5.2 Code Coverage Tool

LLVM-COV GCOV tool [72] was used to get the code coverage information. It analyzes the source code coverage of a program and tests which parts of the code are executed and not executed. It counts how many statements have been executed and gives a percentage of code that has been covered. LLVM-COV GCOV is used with option -b, which gives lines and branches code coverage information. After compiling the programs with code coverage enabled, a file with extension ".gcno" is generated, and after a program

is executed a file with extension ".gcda" is generated. So after testing all the fuzzing files, using Gcov with option -b and "filename.gcno" will obtain the lines and branches code coverage information.

The machine for generating fuzzing files and testing has environment: the CPU is Intel(R)Core(TM) i5-6500 @ 3.20GHZ, memory is 16 GB, Ubuntu 18.04.4 LTS, and OS type 64-bit.

The two types of code coverage lines and branches code coverage were used to obtain and collect the code coverage for the tool. The testing was run to go through all of the programs in the dataset. Some programs have multiple files in the source directory. Each file has a line and branch code coverage. Therefore, we took the averages of each program that has multiple files. In this testing, crashes and code coverage information for each program were collected as shown in Table 6.3. The testing sat to run for a maximum of two days because some programs took more than two days. The testing was planned to keep the program running even after finding a crash. For programs that ran less than one or two days, they stopped after their testing with all fuzzing input files. The experiment was to test one program at a time and it did not start testing the next program until the testing of the previous program was finished. It took about two and half months to finish testing all the programs.

Table 6.3 shows the tool code coverage information for all the programs in the dataset. The table consists of 9 columns: the first column is the program name, the second column is the generation time for generating fuzzing files and it is in minutes and seconds, third column is the number of sample input files, the fourth column is the number of fuzzing files that has been generated, the fifth column is the number of the crash files, the sixth column is the testing time and it is in day, hour, minutes, and second. the seventh column

is indication there is a crash "Y" or no crash "N", the eighth column is lines code coverage, and the ninth is branches code coverage.

For example, in the first row the program name is 3D_Image_Toolkit, the generation time is 3 seconds, the number of sample input files is 1000 files, the number of generated fuzzing files is 6915 files, the number of crashes is 0, and the testing time is 3 and 4 seconds, the crash is no crash, the lines code coverage is 10.41%, and the branches code coverage is 8.75%.

Table 6.3: Tool Code Coverage

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**# of crashed files**	**Testing Time (d:h:m:s)**	**Crash (Y/N)**	**Lines ccov**	**Branches ccov**
3D_Image_Toolkit	0:3	1000	6915	0	0:0:3:4	N	10.41%	8.75%
Accel	0:34	1000	73414	0	0:1:9:11	N	81.35%	95.51%
AIS-Lite	0:1	1000	1000	0	0:0:0:36	N	92.22%	100.00%
anagram_game	0:49	1000	111184	0	0:1:28:6	N	97.45%	100.00%
ASCII_Content_Server	0:2	1000	1091	0	0:0:7:13	N	80.37%	99.04%
ASL6parse	0:7	1000	13268	0	0:0:10:35	N	64.18%	88.61%
Audio_Visualizer	0:24	1000	52502	0	2:0:0:1	N	78.23%	76.46%
Azurad	1:53	1000	196782	55758	0:4:11:57	Y	84.05%	80.64%
Barcoder	0:3	1000	7618	0	0:0:6:24	N	74.42%	73.37%
basic_emulator	2:30	1000	230517	0	0:3:58:49	N	67.06%	70.13%
basic_messaging	0:30	1000	68989	0	0:2:51:56	N	87.29%	97.01%
BIRC	0:22	1000	49426	0	0:0:48:19	N	7.87%	1.86%
BitBlaster	0:2	1000	4059	0	0:0:1:20	N	89.36%	100.00%
Bloomy_Sunday	0:12	1000	30900	1647	0:0:33:30	Y	76.60%	86.46%
Blubber	0:2	1000	1000	0	0:2:46:44	N	37.01%	39.31%
Board_Game	0:29	1000	19342	0	0:0:9:2	N	99.03%	100.00%
BudgIT	0:2	1000	1208	0	0:0:0:28	N	66.61%	77.34%
CableGrind	8:36	1000	300342	0	0:10:37:53	N	31.61%	32.89%
CableGrindLlama	15:40	1000	300932	0	0:11:46:49	N	41.25%	42.73%
Carbonate	0:2	1000	3254	0	0:0:1:47	N	75.71%	92.86%

Continued on next page

Continued from previous page...

Program name	Generation Time (m:s)	# of Sample input files	# of Fuzzing files	# of crashed files	Testing Time (d:h:m:s)	Crash (Y/N)	Lines ccov	Branches ccov
Casino_Games	0:2	1000	4921	0	0:8:29:15	N	80.36%	85.25%
Cereal_Mixup__A_Cereal_Vending_Machine_Controller	0:2	1000	1141	0	0:0:0:42	N	14.03%	22.22%
CGC_Board	0:4	1000	7792	0	0:0:24:14	N	79.94%	87.54%
CGC_File_System	0:2	1000	5235	0	0:0:4:47	N	82.01%	98.73%
CGC_Hangman_Game	0:9	1000	22624	0	0:0:8:52	N	85.00%	88.00%
CGC_Image_Parser	0:19	1000	44454	0	0:0:35:53	N	7.48%	7.89%
CGC_Planet_Markup_Language_Parser	2:45	1000	300100	101	2:0:0:5	Y	75.40%	98.40%
CGC_Symbol_Viewer_CSV	0:37	1000	65962	0	0:0:52:34	N	20.48%	22.14%
CGC_Video_Format_Parser_and_Viewer	0:13	1000	25748	0	0:0:13:26	N	76.81%	94.71%
Character_Statistics	0:2	1000	1652	0	0:0:0:43	N	73.03%	90.32%
Charter	0:2	1000	2458	1	0:0:2:26	Y	89.63%	98.00%
Checkmate	0:2	1000	4995	13	0:2:52:30	Y	95.80%	96.38%
chess_mimic	0:2	1000	1000	0	0:2:46:56	N	55.74%	58.85%
Childs_Game	0:20	1000	49110	0	1:19:45:27	N	91.19%	97.29%
CLOUDCOMPUTE	0:2	1000	2295	0	0:6:22:38	N	83.78%	74.31%
CML	0:25	1000	48607	1029	0:0:47:57	Y	45.56%	56.71%
CNMP	0:2	1000	1000	0	0:0:0:24	N	97.22%	100.00%
COLLIDEOSCOPE	0:4	1000	10957	0	0:2:45:11	N	89.64%	93.10%
commerce_webscale	0:23	1000	54481	0	0:0:41:9	N	16.20%	8.19%
Corinth	0:2	1000	4950	0	0:0:7:11	N	59.78%	73.78%
cotton_swab_arithmetic	0:1	1000	3142	0	0:0:2:8	N	70.87%	89.19%
Cromulence_All_Service	1:57	1000	225262	3	2:0:0:3	Y	64.62%	67.77%
CTTP	0:7	1000	8171	46	0:14:45:44	Y	56.57%	66.77%

Continued on next page

Continued from previous page...

Program name	Generation Time (m:s)	# of Sample input files	# of Fuzzing files	# of crashed files	Testing Time (d:h:m:s)	Crash (Y/N)	Lines ccov	Branches ccov
cyber_blogger	0:13	1000	31935	0	0:10:18:2	N	42.69%	44.99%
DFARS_Sample_Service	0:21	1000	36657	35922	0:2:8:36	Y	54.66%	27.78%
Diary_Parser	0:2	1000	3093	73	0:0:1:25	Y	92.68%	98.63%
Differ	18:13	1000	304261	0	2:0:0:5	N	42.26%	50.06%
Diophantine_Password_Wallet	0:19	1000	19747	23	0:0:6:31	Y	93.48%	94.29%
Dive_Logger	1:27	1000	87386	0	0:2:19:50	N	97.38%	97.16%
Divelogger2	1:20	1000	82757	19578	2:0:0:0	Y	86.77%	91.72%
Document_Rendering_Engine	3:35	1000	222070	3025	0:9:24:51	Y	84.50%	91.67%
Dungeon_Master	6:19	1000	382513	0	0:7:47:48	N	72.81%	78.17%
ECM_TCM_Simulator	0:29	1000	31394	0	0:0:23:26	N	67.44%	84.90%
Eddy	0:2	1000	1226	13	0:1:43:52	Y	92.44%	99.16%
electronictrading	0:2	1000	1006	3	0:1:21:58	Y	75.07%	85.34%
Email_System_2	0:33	999	36647	0	0:12:29:54	N	97.70%	100.00%
Enslavednode_chat	0:53	1000	47641	0	2:0:0:5	N	64.31%	75.19%
Estadio	0:4	1000	4374	0	0:0:1:55	N	54.35%	47.43%
EternalPass	0:6	1000	6984	3982	0:0:11:8	Y	67.88%	67.18%
expression_database	0:24	1000	27119	0	0:2:7:57	N	76.83%	83.37%
FablesReport	0:2	1000	1000	0	0:1:24:41	N	95.21%	100.00%
FaceMag	0:5	1000	5416	0	0:0:3:9	N	7.66%	21.51%
Facilities_Access_Control_System	0:4	1000	4046	0	0:0:2:19	N	58.23%	65.68%
FailAV	0:54	1000	55586	0	0:6:57:59	N	11.36%	7.50%
FASTLANE	0:8	1000	9110	0	0:0:6:3	N	71.34%	71.10%
FileSys	0:14	1000	12929	12	0:0:11:26	Y	72.51%	79.34%
Filesystem_Command_Shell	0:33	1000	37500	143	2:0:0:3	Y	70.26%	85.18%
Finicky_File_Folder	0:13	1000	11366	4772	0:0:13:46	Y	65.74%	67.83%
FISHYXML	0:6	1000	6327	0	0:0:3:50	N	91.27%	99.16%
Flash_File_System	0:5	1000	5164	1	0:0:2:52	Y	65.84%	75.97%

Continued on next page

Continued from previous page...

Program name	Generation Time (m:s)	# of Sample input files	# of Fuzzing files	# of crashed files	Testing Time (d:h:m:s)	Crash (Y/N)	Lines ccov	Branches ccov
Flight_Routes	0:41	1000	46971	7	2:0:0:3	Y	89.30%	96.73%
Fortress	2:24	1000	151818	2	0:2:35:49	Y	90.58%	91.15%
FSK_BBS	28:49	1000	304994	0	1:1:58:20	N	71.09%	91.03%
FSK_Messaging_Service	2:31	1000	127997	0	0:2:40:22	N	67.94%	71.02%
FUN	0:8	1000	9277	0	0:0:6:44	N	20.58%	20.90%
Game_Night	0:12	1000	14618	17	0:0:8:45	Y	81.76%	79.43%
Glue	0:2	1000	1000	0	0:0:0:48	N	92.81%	97.44%
GPS_Tracker	0:3	1000	3096	0	0:0:1:29	N	81.95%	96.17%
GreatView	0:8	1000	5939	0	0:0:37:47	N	70.89%	83.14%
greeter	0:3	1000	3020	0	0:0:1:14	N	71.43%	80.56%
GREYMATTER	5:27	1000	265185	0	0:2:43:43	N	20.45%	20.41%
Gridder	0:2	1000	1000	0	0:0:0:53	N	93.04%	96.15%
Griswold	0:2	1000	1698	0	0:0:0:44	N	17.70%	24.44%
Grit	3:13	1000	147446	520	0:4:55:45	Y	88.34%	97.24%
H20FlowInc	0:11	1000	13484	0	0:0:8:39	N	90.20%	95.77%
HackMan	2:6	1000	135426	3025	0:2:0:57	Y	73.82%	100.00%
hawaii_sets	0:14	1000	15722	0	0:0:6:36	N	62.17%	90.14%
Headscratch	0:2	1000	2000	0	0:0:1:32	N	12.50%	5.88%
HeartThrob	0:3	1000	2987	8	0:0:2:8	Y	98.77%	67.44%
HIGHCOO	0:2	1000	1965	0	0:0:0:43	N	23.81%	50.00%
HighFrequencyTradingAlgo	0:2	1000	1012	8	0:2:36:57	Y	86.62%	91.18%
Hug_Game	0:24	1000	27058	429	0:0:47:15	Y	95.32%	100.00%
humaninterface	0:9	1000	9858	0	0:0:6:9	N	41.96%	58.47%
Image_Compressor	6:36	1000	300056	0	0:9:35:13	N	89.37%	87.50%
INSULATR	1:59	1000	38556	0	0:0:33:57	N	89.47%	98.63%
Kaprica_Go	3:10	1000	124810	0	2:0:0:1	N	92.69%	91.88%
Kaprica_Script_Interpreter	5:47	1000	300707	5439	2:0:0:8	Y	78.89%	87.36%
KKVS	4:56	1000	244941	0	0:5:55:49	N	24.94%	27.22%
KTY_Pretty_Printer	0:37	1000	38319	4	0:0:51:12	Y	82.92%	94.41%
Lazybox	0:26	1000	31622	315	1:18:56:50	Y	61.88%	77.37%

Continued on next page

Continued from previous page...

Program name	Generation Time (m:s)	# of Sample input files	# of Fuzzing files	# of crashed files	Testing Time (d:h:m:s)	Crash (Y/N)	Lines ccov	Branches ccov
LazyCalc	0:6	1000	6120	0	0:0:8:12	N	12.27%	15.62%
LMS	1:57	1000	103545	0	0:0:40:11	N	55.36%	64.29%
Loud_Square_Instant_Messaging_Protocol_LSIMP	0:52	1000	52134	0	0:0:58:32	N	49.08%	33.15%
LulzChat	0:13	1000	14768	0	0:0:10:5	N	6.92%	4.83%
Matchmaker	0:33	1000	35638	771	0:0:18:41	Y	87.10%	87.03%
Material_Temperature_Simulation	0:20	1000	23679	0	2:0:0:2	N	20.29%	24.00%
Mathematical_Solver	0:2	1000	1000	0	0:0:0:29	N	5.54%	7.17%
matrices_for_sale	0:6	1000	6450	4991	0:6:55:42	Y	92.52%	100.00%
Matrix_Math_Calculator	0:23	1000	24499	0	0:0:21:21	N	90.16%	100.00%
Message_Service	1:6	1000	68792	0	0:0:32:56	N	68.84%	78.89%
middleout	0:5	1000	5000	0	0:0:14:30	N	64.50%	90.95%
Minimalistic_Memo_Manager_3M	3:15	1000	197759	0	0:4:4:32	N	98.61%	100.00%
Mixology	0:29	1000	14148	0	0:0:5:49	N	12.00%	16.67%
Modern_Family_Tree	0:4	1000	3090	0	0:0:1:47	N	3.08%	4.62%
Monster_Game	0:44	1000	46249	0	0:0:25:4	N	82.01%	91.51%
Movie_Rental_Service	0:9	1000	9628	156	0:2:59:19	Y	56.09%	60.56%
Movie_Rental_Service_Redux	0:9	1000	9053	0	0:2:9:51	N	58.10%	58.86%
Multi_Arena_Pursuit_Simulator	0:2	1000	1000	0	0:0:0:31	N	96.78%	99.40%
Multi_User_Calendar	0:24	1000	29423	0	0:0:31:6	N	61.93%	74.63%
Multicast_Chat_Server	4:20	968	253273	158444	0:5:48:56	Y	64.22%	68.06%
Multipass	0:1	1000	1260	0	0:0:0:48	N	68.67%	45.58%
Multipass2	0:3	1000	2008	0	0:0:0:39	N	28.98%	32.96%
Multipass3	0:28	1000	24299	25	0:0:9:21	N	55.86%	60.54%

Continued on next page

Continued from previous page...

Program name	Generation Time (m:s)	# of Sample input files	# of Fuzzing files	# of crashed files	Testing Time (d:h:m:s)	Crash (Y/N)	Lines ccov	Branches ccov
Music_Store_Client	0:3	1000	1000	0	0:0:0:36	N	90.94%	100.00%
NarfAgainShell	0:2	1000	1000	0	0:2:46:43	N	29.77%	26.83%
NarfRPN	0:2	1000	1152	0	0:3:12:3	N	78.58%	97.92%
netstorage	0:53	1000	53272	0	0:0:42:43	N	23.03%	19.84%
Network_File_System	0:5	1000	5373	0	0:0:2:33	N	75.93%	100.00%
Network_File_System_v3	0:5	1000	5619	0	0:0:2:57	N	72.60%	79.16%
Network_Queuing_Simulator	0:49	1000	52461	1908	0:0:37:37	Y	93.23%	100.00%
Neural_House	2:30	1000	158947	0	0:6:11:3	N	64.99%	69.41%
No_Paper._Not_Ever._NOPE	0:16	1000	11056	0	0:0:17:5	N	6.73%	8.49%
NoHiC	0:16	1000	16257	0	0:0:12:42	N	15.11%	16.94%
On_Sale	0:2	1000	2165	0	0:6:1:36	N	74.44%	88.42%
One_Amp	0:13	1000	13572	0	0:0:35:17	N	13.78%	17.50%
One_Vote	1:42	1000	109595	0	0:2:31:9	N	66.47%	77.11%
online_job_application	0:31	1000	34581	160	2:0:0:4	Y	66.20%	66.70%
online_job_application2	0:30	1000	34570	1	0:2:13:54	Y	90.92%	100.00%
Order_Up	0:9	1000	9356	0	0:1:25:5	N	3.63%	2.85%
OTPSim	0:51	1000	51325	50428	0:1:27:1	Y	83.60%	87.09%
OUTLAW	0:2	1000	1851	0	0:0:0:50	N	4.81%	2.52%
Overflow_Parking	0:2	1000	1459	0	0:0:5:52	N	82.18%	88.13%
Pac_for_Edges	0:2	1000	1043	0	0:0:0:51	N	96.72%	100.00%
Packet_Analyzer	1:10	1000	67225	0	0:0:24:12	N	7.02%	11.38%
Packet_Receiver	0:28	1000	29017	0	0:0:17:24	N	25.48%	23.54%
Palindrome	0:7	1000	8889	6663	0:0:11:52	Y	82.05%	92.86%
Palindrome2	0:9	1000	9731	7268	0:0:12:56	Y	82.05%	92.86%
Parking_Permit_ManagementSystem_PPMS	0:3	1000	3450	0	0:0:2:19	N	34.48%	35.90%
Particle_Simulator	0:22	1000	25684	0	1:4:36:39	N	94.93%	100.00%

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**# of crashed files**	**Testing Time (d:h:m:s)**	**Crash (Y/N)**	**Lines ccov**	**Branches ccov**
Pattern_Finder	0:12	1000	13553	0	0:0:11:33	N	83.82%	92.32%
payroll	3:34	900	113370	2786	0:1:6:10	Y	90.51%	98.75%
PCM_Message _decoder	5:51	1000	273064	0	0:2:1:11	N	86.09%	97.80%
Personal_Fitness _Manager	0:2	1000	1922	0	0:0:1:1	N	17.77%	6.16%
Pipelined	0:7	1000	7600	0	0:0:2:34	N	15.14%	15.59%
pizza_ordering _system	0:40	1000	46155	709	0:12:49:12	Y	77.59%	88.82%
PKK_Steganography	7:47	1000	300333	0	0:8:8:45	N	7.36%	8.70%
Printer	1:7	1000	72683	103	0:1:19:2	Y	96.92%	99.51%
PRU	0:4	1000	4810	18	0:0:2:13	Y	72.18%	75.00%
PTaaS	1:11	1000	39967	6135	0:2:59:26	Y	79.04%	97.78%
QuadtreeConways	0:6	1000	6228	0	0:5:36:38	N	59.09%	93.38%
Query_Calculator	0:2	1000	1000	41	0:0:0:38	Y	83.45%	95.06%
RAM_based _filesystem	0:2	1000	1848	0	0:0:0:44	N	71.90%	89.05%
reallystream	0:10	1000	10828	0	0:0:3:55	N	71.13%	67.72%
Recipe_and_Pantry _Manager	2:51	1000	186476	2	0:2:3:52	Y	89.91%	100.00%
Recipe_Database	3:15	1000	196705	680	2:0:0:7	Y	83.54%	89.48%
REDPILL	0:2	1000	1254	0	0:3:4:42	N	56.52%	63.59%
Rejistar	0:2	1000	1000	0	0:0:0:32	N	78.05%	87.02%
REMATCH_1–Hat _Trick–Morris _Worm	0:3	1000	3176	0	0:0:1:19	N	15.64%	15.71%
REMATCH_2–Mail _Server–Crackaddr	0:3	1000	4003	28	0:0:1:50	Y	87.38%	88.59%
REMATCH_3–Address_Resolution_Service–SQL_Slammer	0:6	1000	6569	286	0:0:3:29	Y	69.64%	79.53%
REMATCH_4–CG-CRPC_Server–MS08-067	0:2	1000	2947	618	0:0:2:21	Y	40.59%	37.66%

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**# of crashed files**	**Testing Time (d:h:m:s)**	**Crash (Y/N)**	**Lines ccov**	**Branches ccov**
REMATCH_5–File_Explorer–LNK_Bug	0:9	1000	9942	3	0:0:4:35	Y	69.64%	82.05%
REMATCH_6–Secure_Server–Heartbleed	0:12	1000	11409	5734	0:0:20:59	Y	14.31%	12.76%
Resort_Modeller	0:4	1000	2972	0	0:0:1:48	N	11.11%	16.67%
root64_and_parcour	0:13	1000	13901	0	0:0:8:33	N	89.51%	95.56%
router_simulator	11:46	1000	165507	0	0:2:12:42	N	46.35%	98.58%
RRPN	0:2	1000	1907	111	0:0:12:18	Y	96.35%	100.00%
Sad_Face_Template_Engine_SFTE	0:19	1000	23031	45	0:0:21:44	Y	78.64%	86.83%
Sample_Shipgame	0:45	1000	51385	28509	0:0:58:17	Y	95.10%	100.00%
SAuth	0:5	1000	5799	0	0:0:4:26	N	62.70%	71.43%
Scrum_Database	0:2	1000	1202	0	0:0:2:7	N	60.18%	66.52%
SCUBA_Dive_Logging	1:18	1000	81656	32695	2:0:0:7	Y	93.31%	100.00%
Secure_Compression	0:42	1000	46846	44827	0:1:36:10	Y	96.72%	98.25%
Sensr	0:16	1000	15850	0	0:0:12:25	N	98.14%	96.36%
SFTSCBSISS	0:3	1000	2218	0	0:0:0:52	N	75.74%	92.71%
Shipgame	0:4	1000	3998	0	0:0:1:26	N	9.35%	6.95%
Shortest_Path_Tree_Calculator	0:12	1000	13000	0	0:17:18:56	N	61.91%	69.25%
ShoutCTF	0:31	1000	35157	0	0:0:29:11	N	9.10%	6.39%
SIGSEGV	0:2	1000	1006	0	0:0:2:26	N	66.97%	52.17%
simple_integer_calculator	0:2	1000	2136	24	0:4:41:45	Y	94.51%	100.00%
Simple_Stack_Machine	0:3	1000	3539	0	0:0:1:2	N	37.93%	31.75%
simplenote	4:32	1000	300072	7399	0:12:19:16	Y	74.32%	82.10%
simpleOCR	0:12	1000	13604	0	0:3:11:37	N	90.40%	91.55%
Single-Sign-On	0:8	1000	6446	0	0:0:30:43	N	65.32%	72.73%
SLUR_reference_implementation	0:2	1000	1000	0	0:0:4:54	N	50.25%	99.42%

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**# of crashed files**	**Testing Time (d:h:m:s)**	**Crash (Y/N)**	**Lines ccov**	**Branches ccov**
Snail_Mail	0:2	1000	1375	0	0:3:49:38	N	70.14%	80.35%
SOLFEDGE	0:3	1000	1491	0	0:0:0:30	N	97.54%	99.06%
Sorter	0:8	1000	1028	0	0:0:0:47	N	44.51%	15.09%
Space_Attackers	0:41	1000	46057	0	0:0:40:53	N	81.50%	81.68%
SPIFFS	0:25	1000	25222	19	0:6:6:59	Y	84.44%	91.70%
Square_Rabbit	0:10	1000	12261	0	0:13:47:32	N	84.44%	88.89%
stack_vm	0:4	1000	1000	0	0:0:0:24	N	75.10%	82.81%
Stock_Exchange_Simulator	0:58	1000	60834	0	0:0:22:4	N	67.04%	56.44%
stream_vm	5:18	1000	300132	169	0:22:12:48	Y	91.25%	97.44%
stream_vm2	5:44	1000	300202	0	0:7:4:49	N	84.46%	94.06%
Street_map_service	0:4	1000	4205	0	0:0:3:35	N	64.14%	55.94%
String_Info_Calculator	1:22	1000	88192	0	0:0:38:2	N	84.38%	100.00%
String_Storage_and_Retrieval	0:33	1000	33926	0	0:2:5:3	N	24.06%	35.38%
TAINTEDLOVE	1:10	1000	69188	16100	1:6:57:38	Y	61.84%	66.67%
Tennis_Ball_Motion_Calculator	0:7	1000	9134	23	0:0:3:7	Y	93.06%	100.00%
Terrible_Ticket_Tracker	0:4	1000	4439	0	0:12:23:47	N	31.72%	23.40%
TextSearch	3:50	1000	242852	1	0:13:28:1	Y	88.71%	93.96%
TFTTP	0:2	1000	1186	0	0:0:0:36	N	74.14%	69.61%
The_Longest_Road	0:45	1000	48838	784	0:0:22:19	Y	93.72%	99.71%
Thermal_Controller_v2	0:13	1000	14557	0	0:0:7:49	N	21.41%	9.87%
Thermal_Controller_v3	0:17	1000	19253	0	0:0:11:26	N	13.34%	5.36%
TIACA	0:6	1000	6620	0	0:0:2:35	N	11.85%	20.83%
Tick-A-Tack	0:2	1000	1000	0	0:0:0:19	N	90.29%	100.00%
tribute	6:49	1000	253978	0	0:1:33:49	N	29.89%	29.17%
TVS	5:44	1000	300653	1	0:10:23:41	Y	83.27%	83.33%
university_enrollment	0:25	1000	27849	425	0:17:0:8	Y	79.64%	95.25%

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**# of crashed files**	**Testing Time (d:h:m:s)**	**Crash (Y/N)**	**Lines ccov**	**Branches ccov**
User_Manager	0:12	1000	14054	0	1:2:15:14	N	81.64%	69.03%
UTF-late	0:2	1000	1000	0	0:2:46:43	N	85.53%	100.00%
ValveChecks	0:2	1000	1767	0	0:0:0:34	N	11.45%	14.64%
Vector_Graphics_2	0:2	1000	2377	2	0:0:0:59	Y	87.74%	93.83%
Vector_Graphics _Format	0:3	1000	2371	14	0:0:0:58	Y	75.92%	83.69%
Venture_Calculator	0:29	1000	32499	0	0:0:26:22	N	90.32%	89.84%
vFilter	6:49	1000	300164	0	0:8:22:39	N	54.09%	57.07%
Virtual_Machine	0:35	1000	35840	0	0:0:23:42	N	15.17%	18.00%
virtual_pet	0:4	1000	4445	3	0:0:1:35	Y	91.21%	100.00%
Water_Treatment _Facility_Simulator	0:55	1000	59185	1	0:0:49:21	Y	92.31%	96.93%
WhackJack	0:49	1000	55939	9042	0:13:27:17	Y	91.79%	94.78%
WordCompletion	0:3	1000	3926	1	0:0:1:6	Y	85.00%	91.67%
XStore	1:21	1000	78319	0	0:1:11:51	N	3.15%	2.75%
yolodex	0:9	1000	11061	0	0:0:4:31	N	94.56%	100.00%

After the experiment, the code coverage information showed how much of each program had executed. Two types of code coverage are shown in the table, which are line coverage and branch coverage. We found the average of line code coverage is about 65%. Also, we found the average of branch code coverage is about 71%. They tell us that the generated fuzzing files had pretty good coverage. Also, we used this code coverage information to compare our tool code coverage with sample input code coverage in Section 6.5.4.

6.5.3 Low Code Coverage Programs

There are some programs with low code coverage. In general, low code coverage is mainly caused when a program has user input that is not compatible with the programs requirement and the commands that come after entering the user input cannot be

executed correctly. Some programs cannot be executed correctly in our system because of uninitialized variables. For example, 3D_Image_Toolkit and XStore are two programs that have lower code coverage.

Table 6.4: 3D_Image_toolkit Code Coverage

3D_Image_toolkit		
File name	**Line code coverage**	**branches code coverage**
3dc.c	18.37%	20.00%
compress.c	0.00%	0.00%
main.c	12.87%	6.25%

After the investigation, 3D_Image_Toolkit program has low code coverage in lines and branches code coverage. 3D_Image_Toolkit has three C code files as shown in Table 6.4. In main.c, (part of the code is shown in Listing 6.3) a variable *choice* is declared but never get a value. Then a while loop exist later in the main.c file. The evaluation of the loop is based on the value of *choice*. In our system, apparently the variable has a value of 0, and the evaluation of the while loop is FALSE. Hence the whole while loop which takes more than 50 lines of executable code are not executed at all. This leads to low code coverage on the main.c file.

The Listings 6.3 and 6.4 show the covered and uncovered lines from the GCOV coverage files. There are many lines which have numbers, (-), and (#####). So, if the beginning of the line has (-), that indicates it is not an executable line; if the line starts by a number, this number tells the lines were executed (covered) and it tells how many times this line was executed, and if a line starts by (#####), the line was not executed (uncovered).

In Listing 6.3, the main function has the function cgc_menu(). In cgc_menu() the input "choice" variable (declared on line 56) never got a value but used in the while loop on line 81. From the coverage information, we can tell the while loop was checked, but the loop body was not executed. This indicates the condition of the while loop is always evaluated to False. By skipping the while loop, it affects the code coverage of the program.

In addition, on line 84, the switch-case statement has (#####) at the beginning of the line which means the line was not executed. The switch cases in the switch statement call functions in 3dc.c and compress.c files. Because of not executing the switch statement, many functions are not called. So Table 6.4 shows that other C files have low line and branch coverage or even 0. Therefore, 3D_Image_toolkit overall code coverage is lower.

The total number of executable lines in main.c file is 101. There are 13 executed lines and the uncovered lines are 88 lines. To calculate the percentage of covered lines we can divide the covered lines number by the total lines number so 13/101 = 12.87%.

Listing 6.3: 3D_Image_toolkit Program

```
        ....
      -:   55:void cgc_menu() {
      -:   56:  char choice;
      -:   57:  char coord;
      -:   58:  uint16_t val;
      -:   59:
      -:   60:  char new_init[4096];
 113136:   61:  t3DCPixel *last_shown = NULL;
      -:   62:
      -:   63:#ifdef PATCHED_1
      -:   64:  uint8_t *out_data =
    cgc_malloc(MAX_FILE_SIZE);
      -:   65:  uint8_t *in_data =
    cgc_malloc(MAX_FILE_SIZE);
      -:   66:#else
      -:   67:  uint8_t out_data[3072];
      -:   68:  uint8_t in_data[3072];
      -:   69:#endif
      -:   70:
```

```
    56568:    71:  uint16_t compress_len = 0;
    56568:    72:  uint8_t decompress_flag = 0;
        -:    73:
    56568:    74:  uint16_t pixelCount =
    MAGIC_PAGE_SIZE / sizeof(t3DCPixel);
        -:    75:
    56568:    76:  t3DCPixel **px_list =
    cgc_malloc(pixelCount * sizeof(t3DCPixel*));
    56568:    77:  cgc_memset(px_list, 0,
    pixelCount * sizeof(t3DCPixel*));
        -:    78:
    56568:    79:  cgc_ReadFile(px_list);
        -:    80:
   113136:    81:  while(choice) {
    #####:    82:    cgc_receive_bytes(&choice, 1);
        -:    83:
    #####:    84:    switch(choice) {
          ....
        -:   227:    }
        -:   228:  }
    56568:   229:  cgc_free(px_list);
   113136:   230:}
        -:   231:
        -:   232:int main(int cgc_argc, char *cgc_argv[]) {
        -:   233:
    56568:   234:    cgc_printf("3D Coordinates
              (3DC) Image File Format Tools\n");
        -:   235:
    56568:   236:    cgc_menu();
        -:   237:
    56568:   238:    return 0;
        -:   239:}
```

XStore program has low code coverage in lines and branches. It consists of three C code files. As shown in Table 6.5, there is low code coverage in XStore as well.

Table 6.5: XStore Code Coverage

XStore		
File name	**Line code coverage**	**branches code coverage**
service.c	7.07%	7.36%
tr.c	0.00%	0.00%
xpack.c	2.39%	0.90%

Because after entering user input, the commands that came later cannot be executed. The reason for this is that unknown byte values were not understood or not compatible with the program requirement. Moreover, the code overage of service.c is low, which causes the other C code files to have lower code coverage, which affects the average of lines and branches code coverage.

From examining the code coverage report, shown in Listing 6.4, the cases in the switch-case statement (started from line 717) were not executed except the default case (on line 739). The variable "command" was read in from function cgc_fread (on line 708). It means the user input was not read in properly or it was not in the correct format. Therefore, none of the switch statements were covered, and many function calls (functions in tr.c and xpack.c files) were not executed. It led to a low code coverage on the program.

If counting the executed lines and not executed lines, in service.c file, there were 26 covered lines and the uncovered lines are 342 lines. Therefore, the total number of lines is 368. To calculate the percentage of covered lines, we can divide the covered lines number by the total line number, 26/368 = 7.07%.

Listing 6.4: XStore Program

```
    ....
        -:  692:int main(int secret_page_i, char *unused[])
```

```
        {
        -:  693:    secret_page_i = CGC_FLAG_PAGE_ADDRESS;
        -:  694:
        -:  695:    void *secret_page =(void *)secret_page_i;
    78712:  696:    uint64_t command;
    39356:  697:    cgc_size_t size;
    39356:  698:    uint8_t debug = 0;
        -:  699:
    39356:  700:    cgc_g_ctx = cgc_xpk_init(1024);
        -:  701:
    39356:  702:    cgc_check_seed();
        -:  703:
    39356:  704:    cgc_fbuffered(cgc_stdout, 1);
  5881291:  705:    while (1)
        -:  706:    {
177282850:  707:        cgc_fflush(cgc_stdout);
177282850:  708:        if (cgc_fread(&command,
                          sizeof(uint64_t),
                          cgc_stdin) != sizeof(uint64_t))
        -:  709:            break;
177256503:  710:        if (cgc_fread(&size,
                            sizeof(cgc_size_t), cgc_stdin) !=
                         sizeof(cgc_size_t))
        -:  711:            break;
177243494:  712:        if (size > MAX_DATA_LEN)
        -:  713:        {
171401559:  714:            cgc_printf("Wrong." NL);
171401559:  715:            continue;
        -:  716:        }
  5841935:  717:switch (command)
        -:  718:        {
        -:  719:            case CMD_STORE:
    #####:  720:                cgc_handle_store(size);
    #####:  721:                break;
        -:  722:            case CMD_LOOKUP:
    #####:  723:                cgc_handle_lookup(size);
    #####:  724:                break;
        -:  725:            case CMD_DELETE:
    #####:  726:                cgc_handle_delete(size);
    #####:  727:                break;
        -:  728:            case CMD_DEBUG:
    #####:  729:                cgc_handle_debug(size, &debug);
    #####:  730:                break;
        -:  731:            case CMD_PRINT:
```

```
    #####:  732:                cgc_handle_print(size, debug);
    #####:  733:                    break;
        -:  734:                case CMD_QUIT:
    #####:  735:                    cgc_send_response(RES_OK);
    #####:  736:                    cgc_fflush(cgc_stdout);
    #####:  737:                    cgc_exit(0);
        -:  738:                default:
  5841935:  739:                  cgc_send_response(RES_INVALID);
  5841935:  740:                    break;
        -:  741:            }
        -:  742:    }
    39356:  743:    cgc_fflush(cgc_stdout);
        -:  744:
    39356:  745:    return 0;
    39356:  746:}
```

6.5.4 Code Coverage Difference Between Generated Fuzzing Files and Sample Inputs Files

The Table 6.6 shows the comparison between our regular fuzzer coverage and sample input files code coverage. Our regular fuzzer uses 10% percentage which means 10% of number of sample input files that our tool used to obtain the grammars. It shows that the code coverage of our fuzzer is similar to the sample input files' code coverage. The Table 6.6 has four major columns: the first column is the program name; the second column is the code coverage for our regular fuzzer. It has two columns lines and branches code coverage; the third column is sample input files' code coverage and it has two column which are lines and branches code coverage; and the fourth column is the difference which has the calculated difference between our tool's code coverage and sample input files' code coverage. The difference column has two column: lines and branches differences.

If the difference number is 0.0, it means the code coverage has the same coverage between our regular fuzzer code coverage and sample code coverage. If the difference is a positive number, it means that our regular fuzzer has increased code coverage for a

particular program. If the difference number is negative, it means our regular fuzzer code coverage has lower code coverage.

Looking at column difference Table 6.6 we can see most of the programs have 0.0 numbers difference numbers for lines and branches columns. Therefore, it indicates that the tool has similar code coverage with sample inputs code coverage.

For example, in the first row is 3D_Image_Toolkit, 10% tool lines code coverage is 10.41% and the branches code coverage is 8.75%; the sample line code coverage is 10.41% and branches code coverage is 8.75%; and the difference between them is 0.00% for lines and branches code coverage.

Table 6.6: Code Coverage Differences

Program Name	**Our Fuzzer CCOV**		**Samples CCOV**		**Difference**	
	Lines ccov	**Branches ccov**	**Lines ccov**	**Branches ccov**	**Lines**	**Branchs**
3D_Image_Toolkit	10.41%	8.75%	10.41%	8.75%	0.00%	0.00%
Accel	81.13%	95.51%	81.07%	95.51%	0.06%	0.00%
AIS-Lite	92.22%	100.00%	92.22%	100.00%	0.00%	0.00%
anagram_game	97.11%	99.55%	97.05%	99.33%	0.06%	0.22%
ASCII_Content_Server	80.37%	99.04%	80.37%	99.04%	0.00%	0.00%
ASL6parse	71.49%	96.41%	73.58%	98.73%	-2.09%	-2.32%
Audio_Visualizer	79.88%	78.88%	78.09%	76.46%	1.79%	2.42%
Azurad	82.41%	78.94%	82.29%	78.42%	0.12%	0.52%
Barcoder	74.65%	74.24%	74.76%	74.68%	-0.11%	-0.44%
basic_emulator	66.56%	69.60%	66.83%	69.81%	-0.27%	-0.21%
basic_messaging	87.29%	97.01%	87.29%	97.01%	0.00%	0.00%
BIRC	48.23%	46.65%	48.23%	46.65%	0.00%	0.00%
BitBlaster	89.36%	100.00%	89.36%	100.00%	0.00%	0.00%
Bloomy_Sunday	76.60%	86.46%	76.60%	86.46%	0.00%	0.00%
Blubber	36.99%	39.27%	37.01%	39.31%	-0.02%	-0.04%
Board_Game	99.03%	100.00%	99.03%	100.00%	0.00%	0.00%
BudgIT	68.09%	77.34%	68.09%	77.34%	0.00%	0.00%
CableGrind	30.78%	32.89%	30.36%	32.89%	0.42%	0.00%
CableGrindLlama	40.87%	42.73%	40.11%	42.73%	0.76%	0.00%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
Carbonate	75.71%	92.86%	75.71%	92.86%	0.00%	0.00%
Casino_Games	73.77%	81.87%	72.13%	81.73%	1.64%	0.14%
Cereal_Mixup__A _Cereal_Vending_Machine_Controller	59.81%	66.93%	58.81%	65.94%	1.00%	0.99%
CGC_Board	79.88%	88.74%	79.94%	89.08%	-0.06%	-0.34%
CGC_File_System	83.39%	98.73%	83.39%	98.73%	0.00%	0.00%
CGC_Hangman _Game	85.00%	88.00%	85.00%	88.00%	0.00%	0.00%
CGC_Image_Parser	44.25%	49.18%	42.02%	44.93%	2.23%	4.25%
CGC_Planet_Markup _Language_Parser	73.29%	98.06%	73.58%	98.14%	-0.29%	-0.08%
CGC_Symbol_Viewer _CSV	21.15%	22.14%	21.05%	22.14%	0.10%	0.00%
CGC_Video_Format _Parser_and_Viewer	77.01%	94.71%	77.27%	94.71%	-0.26%	0.00%
Character_Statistics	73.03%	90.32%	73.03%	90.32%	0.00%	0.00%
Charter	89.77%	98.67%	90.06%	100.00%	-0.29%	-1.33%
Checkmate	94.98%	95.78%	94.98%	95.78%	0.00%	0.00%
chess_mimic	55.74%	58.85%	55.74%	58.85%	0.00%	0.00%
Childs_Game	90.00%	97.29%	90.35%	97.29%	-0.35%	0.00%
CLOUDCOMPUTE	83.20%	73.81%	82.78%	73.81%	0.42%	0.00%
CML	45.57%	56.75%	45.62%	56.77%	-0.05%	-0.02%
CNMP	97.22%	100.00%	97.22%	100.00%	0.00%	0.00%
COLLIDEOSCOPE	88.67%	92.91%	88.16%	92.23%	0.51%	0.68%
commerce_webscale	34.63%	38.68%	35.17%	39.37%	-0.54%	-0.69%
Corinth	65.11%	78.22%	65.11%	78.22%	0.00%	0.00%
cotton_swab_arithmetic	84.51%	98.20%	86.61%	100.00%	-2.10%	-1.80%
Cromulence_All _Service	41.80%	37.03%	41.22%	36.34%	0.58%	0.69%
CTTP	56.57%	66.77%	56.57%	66.77%	0.00%	0.00%
cyber_blogger	41.95%	43.34%	41.87%	43.34%	0.08%	0.00%
DFARS_Sample _Service	53.67%	27.16%	54.10%	27.78%	-0.43%	-0.62%
Diary_Parser	93.50%	98.63%	93.50%	98.63%	0.00%	0.00%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
Differ	35.56%	42.91%	35.56%	42.91%	0.00%	0.00%
Diophantine_Password_Wallet	93.48%	94.29%	93.48%	94.29%	0.00%	0.00%
Dive_Logger	97.38%	97.16%	97.38%	97.16%	0.00%	0.00%
Divelogger2	86.55%	91.72%	86.51%	91.72%	0.04%	0.00%
Document_Rendering_Engine	83.92%	91.67%	84.00%	91.67%	-0.08%	0.00%
Dungeon_Master	69.84%	74.75%	70.78%	75.84%	-0.94%	-1.09%
ECM_TCM_Simulator	67.76%	85.17%	67.64%	84.90%	0.12%	0.27%
Eddy	92.44%	99.16%	92.35%	99.16%	0.09%	0.00%
electronictrading	73.86%	85.33%	74.84%	86.66%	-0.98%	-1.33%
Email_System_2	91.87%	99.57%	92.27%	100.00%	-0.40%	-0.43%
Enslavednode_chat	76.65%	82.64%	77.10%	82.73%	-0.45%	-0.09%
Estadio	51.61%	44.61%	54.35%	47.43%	-2.74%	-2.82%
EternalPass	87.29%	86.54%	89.65%	88.82%	-2.36%	-2.28%
expression_database	76.14%	82.68%	76.16%	82.89%	-0.02%	-0.21%
FablesReport	95.21%	100.00%	95.21%	100.00%	0.00%	0.00%
FaceMag	35.57%	47.01%	35.57%	47.01%	0.00%	0.00%
Facilities_Access_Control_System	59.60%	65.68%	59.81%	65.68%	-0.21%	0.00%
FailAV	26.91%	24.82%	26.91%	24.82%	0.00%	0.00%
FASTLANE	71.34%	71.10%	71.34%	71.10%	0.00%	0.00%
FileSys	72.51%	79.34%	72.51%	79.34%	0.00%	0.00%
Filesystem_Command_Shell	77.68%	89.01%	76.81%	88.61%	0.87%	0.40%
Finicky_File_Folder	66.34%	68.06%	66.14%	67.83%	0.20%	0.23%
FISHYXML	91.41%	99.30%	91.41%	99.37%	0.00%	-0.07%
Flash_File_System	67.72%	79.05%	69.10%	81.34%	-1.38%	-2.29%
Flight_Routes	81.78%	94.06%	81.41%	93.77%	0.37%	0.29%
Fortress	90.55%	91.15%	90.55%	91.15%	0.00%	0.00%
FSK_BBS	71.09%	91.03%	71.09%	91.03%	0.00%	0.00%
FSK_Messaging_Service	67.94%	71.02%	67.94%	71.02%	0.00%	0.00%
FUN	20.58%	20.90%	20.58%	20.90%	0.00%	0.00%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
Game_Night	82.35%	79.43%	82.35%	79.43%	0.00%	0.00%
Glue	92.81%	97.44%	92.81%	97.44%	0.00%	0.00%
GPS_Tracker	82.61%	97.45%	82.53%	97.45%	0.08%	0.00%
GreatView	70.89%	83.14%	70.89%	83.14%	0.00%	0.00%
greeter	71.43%	80.56%	71.43%	80.56%	0.00%	0.00%
GREYMATTER	20.45%	20.41%	20.45%	20.41%	0.00%	0.00%
Gridder	93.04%	96.15%	93.04%	96.15%	0.00%	0.00%
Griswold	36.66%	44.33%	36.66%	44.33%	0.00%	0.00%
Grit	88.25%	97.18%	88.32%	97.24%	-0.07%	-0.06%
H20FlowInc	91.75%	97.95%	91.75%	97.95%	0.00%	0.00%
HackMan	73.69%	100.00%	73.82%	100.00%	-0.13%	0.00%
hawaii_sets	63.53%	90.97%	63.84%	90.97%	-0.31%	0.00%
Headscratch	23.50%	20.88%	23.50%	20.88%	0.00%	0.00%
HeartThrob	98.77%	67.49%	98.77%	67.50%	0.00%	-0.01%
HIGHCOO	23.81%	50.00%	23.81%	50.00%	0.00%	0.00%
HighFrequencyTradingAlgo	84.98%	88.24%	85.21%	88.24%	-0.23%	0.00%
Hug_Game	95.49%	100.00%	94.72%	100.00%	0.77%	0.00%
humaninterface	41.96%	58.85%	41.96%	59.62%	0.00%	-0.77%
Image_Compressor	89.37%	87.50%	89.37%	87.50%	0.00%	0.00%
INSULATR	89.47%	98.63%	89.47%	98.63%	0.00%	0.00%
Kaprica_Go	92.69%	91.88%	92.69%	91.88%	0.00%	0.00%
Kaprica_Script_Interpreter	75.92%	84.67%	75.74%	85.40%	0.18%	-0.73%
KKVS	49.89%	54.45%	49.89%	54.45%	0.00%	0.00%
KTY_Pretty_Printer	82.75%	94.41%	82.74%	94.41%	0.01%	0.00%
Lazybox	48.68%	58.09%	48.69%	57.88%	-0.01%	0.21%
LazyCalc	23.87%	32.20%	23.87%	32.20%	0.00%	0.00%
LMS	55.66%	64.88%	55.36%	64.29%	0.30%	0.59%
Loud_Square_Instant_Messaging_Protocol_LSIMP	50.17%	33.15%	50.72%	33.15%	-0.55%	0.00%
LulzChat	6.92%	4.83%	6.92%	4.83%	0.00%	0.00%
Matchmaker	87.10%	87.03%	87.10%	87.03%	0.00%	0.00%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
Material_Temperature_Simulation	64.01%	67.50%	64.01%	67.50%	0.00%	0.00%
Mathematical_Solver	61.75%	75.56%	61.87%	77.62%	-0.12%	-2.06%
matrices_for_sale	91.90%	100.00%	90.65%	100.00%	1.25%	0.00%
Matrix_Math_Calculator	90.31%	100.00%	90.38%	100.00%	-0.07%	0.00%
Message_Service	68.84%	78.89%	68.84%	78.89%	0.00%	0.00%
middleout	64.78%	90.95%	64.91%	90.95%	-0.13%	0.00%
Minimalistic_Memo_Manager_3M	97.48%	100.00%	97.52%	100.00%	-0.04%	0.00%
Mixology	19.27%	23.67%	19.27%	23.67%	0.00%	0.00%
Modern_Family_Tree	43.09%	47.12%	43.09%	47.12%	0.00%	0.00%
Monster_Game	81.37%	90.48%	84.24%	93.05%	-2.87%	-2.57%
Movie_Rental_Service	56.06%	60.56%	56.09%	60.56%	-0.03%	0.00%
Movie_Rental_Service_Redux	58.10%	58.86%	58.10%	58.86%	0.00%	0.00%
Multi_Arena_Pursuit_Simulator	96.78%	99.40%	96.78%	99.40%	0.00%	0.00%
Multi_User_Calendar	66.72%	81.52%	66.83%	82.05%	-0.11%	-0.53%
Multicast_Chat_Server	64.22%	68.06%	64.22%	68.06%	0.00%	0.00%
Multipass	68.99%	45.58%	68.67%	45.58%	0.32%	0.00%
Multipass2	32.94%	39.16%	32.71%	38.73%	0.23%	0.43%
Multipass3	56.79%	61.14%	55.98%	60.44%	0.81%	0.70%
Music_Store_Client	90.94%	100.00%	90.94%	100.00%	0.00%	0.00%
NarfAgainShell	48.59%	53.64%	48.32%	53.49%	0.27%	0.15%
NarfRPN	82.33%	98.92%	82.33%	98.92%	0.00%	0.00%
netstorage	35.11%	40.10%	35.11%	40.10%	0.00%	0.00%
Network_File_System	76.72%	100.00%	76.72%	100.00%	0.00%	0.00%
Network_File_System_v3	74.94%	82.80%	74.54%	82.13%	0.40%	0.67%
Network_Queuing_Simulator	93.23%	100.00%	93.23%	100.00%	0.00%	0.00%
Neural_House	64.99%	69.41%	64.99%	69.41%	0.00%	0.00%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
No_Paper._Not_Ever._NOPE	24.73%	29.22%	24.73%	29.22%	0.00%	0.00%
NoHiC	26.84%	29.77%	26.84%	29.77%	0.00%	0.00%
On_Sale	74.14%	87.92%	73.69%	87.09%	0.45%	0.83%
One_Amp	15.27%	19.61%	15.71%	20.15%	-0.44%	-0.54%
One_Vote	67.29%	77.20%	66.47%	77.11%	0.82%	0.09%
online_job_application	57.56%	61.64%	57.37%	61.02%	0.19%	0.62%
online_job_application2	90.27%	99.43%	90.92%	100.00%	-0.65%	-0.57%
Order_Up	3.63%	2.85%	3.63%	2.85%	0.00%	0.00%
OTPSim	78.76%	79.03%	78.76%	79.03%	0.00%	0.00%
OUTLAW	4.81%	2.52%	4.81%	2.52%	0.00%	0.00%
Overflow_Parking	82.18%	88.13%	82.18%	88.13%	0.00%	0.00%
Pac_for_Edges	96.72%	100.00%	96.72%	100.00%	0.00%	0.00%
Packet_Analyzer	7.02%	11.38%	7.02%	11.38%	0.00%	0.00%
Packet_Receiver	25.48%	23.54%	25.48%	23.54%	0.00%	0.00%
Palindrome	82.05%	92.86%	82.05%	92.86%	0.00%	0.00%
Palindrome2	82.05%	92.86%	82.05%	92.86%	0.00%	0.00%
Parking_Permit_Management_System_PPMS	37.02%	37.95%	38.29%	38.97%	-1.27%	-1.02%
Particle_Simulator	94.48%	100.00%	94.42%	100.00%	0.06%	0.00%
Pattern_Finder	83.82%	92.32%	83.82%	92.32%	0.00%	0.00%
payroll	91.33%	98.75%	92.98%	98.75%	-1.65%	0.00%
PCM_Message_decoder	73.22%	82.60%	73.29%	82.35%	-0.07%	0.25%
Personal_Fitness_Manager	17.77%	6.16%	17.77%	6.16%	0.00%	0.00%
Pipelined	15.14%	15.59%	15.14%	15.59%	0.00%	0.00%
pizza_ordering_system	77.59%	88.82%	77.59%	88.82%	0.00%	0.00%
PKK_Steganography	7.36%	8.70%	7.36%	8.70%	0.00%	0.00%
Printer	96.71%	99.51%	96.60%	99.51%	0.11%	0.00%
PRU	74.96%	76.39%	77.56%	79.17%	-2.60%	-2.78%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
PTaaS	79.04%	97.78%	79.04%	97.78%	0.00%	0.00%
QuadtreeConways	58.96%	92.36%	59.09%	92.46%	-0.13%	-0.10%
Query_Calculator	83.45%	95.06%	83.45%	95.06%	0.00%	0.00%
RAM_based_filesystem	73.68%	90.64%	73.68%	90.64%	0.00%	0.00%
reallystream	72.03%	68.25%	72.03%	68.25%	0.00%	0.00%
Recipe_and_Pantry_Manager	89.14%	100.00%	89.34%	100.00%	-0.20%	0.00%
Recipe_Database	80.48%	85.99%	79.67%	84.72%	0.81%	1.27%
REDPILL	52.42%	58.15%	52.90%	60.33%	-0.48%	-2.18%
Rejistar	78.59%	87.02%	78.59%	87.02%	0.00%	0.00%
REMATCH_1–Hat_Trick–Morris _Worm	15.64%	15.71%	15.64%	15.71%	0.00%	0.00%
REMATCH_2–Mail_Server–Crackaddr	87.38%	88.59%	87.38%	88.59%	0.00%	0.00%
REMATCH_3–Address_Resolution_Service–SQL_Slammer	69.82%	79.53%	70.17%	79.53%	-0.35%	0.00%
REMATCH_4–CGC-RPC_Server–MS08-067	46.11%	47.34%	46.28%	47.42%	-0.17%	-0.08%
REMATCH_5–File_Explorer–LNK_Bug	72.20%	82.91%	72.35%	82.91%	-0.15%	0.00%
REMATCH_6–Secure_Server–Heartbleed	14.60%	12.76%	14.60%	12.76%	0.00%	0.00%
Resort_Modeller	11.11%	16.67%	11.11%	16.67%	0.00%	0.00%
root64_and_parcour	89.51%	95.56%	89.51%	95.56%	0.00%	0.00%
router_simulator	46.26%	98.58%	46.22%	98.58%	0.04%	0.00%
RRPN	96.35%	100.00%	96.35%	100.00%	0.00%	0.00%
Sad_Face_Template_Engine_SFTE	78.64%	86.83%	78.64%	86.83%	0.00%	0.00%
Sample_Shipgame	93.95%	100.00%	93.66%	100.00%	0.29%	0.00%
SAuth	62.70%	71.43%	62.70%	71.43%	0.00%	0.00%
Scrum_Database	37.20%	31.06%	37.20%	31.06%	0.00%	0.00%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
SCUBA_Dive_Logging	78.23%	86.95%	78.89%	87.29%	-0.66%	-0.34%
Secure_Compression	96.58%	98.25%	96.29%	98.25%	0.29%	0.00%
Sensr	98.14%	96.36%	98.14%	96.36%	0.00%	0.00%
SFTSCBSISS	77.02%	93.75%	77.02%	93.75%	0.00%	0.00%
Shipgame	9.35%	6.95%	9.35%	6.95%	0.00%	0.00%
Shortest_Path_Tree_Calculator	59.05%	65.55%	56.40%	62.94%	2.65%	2.61%
ShoutCTF	9.10%	6.39%	9.10%	6.39%	0.00%	0.00%
SIGSEGV	66.97%	52.17%	66.97%	52.17%	0.00%	0.00%
simple_integer_calculator	92.01%	99.10%	91.28%	98.78%	0.73%	0.32%
Simple_Stack_Machine	37.93%	31.75%	37.93%	31.75%	0.00%	0.00%
simplenote	74.32%	82.10%	74.32%	82.10%	0.00%	0.00%
simpleOCR	91.72%	92.49%	90.68%	91.55%	1.04%	0.94%
Single-Sign-On	65.32%	72.73%	65.32%	72.73%	0.00%	0.00%
SLUR_reference_implementation	50.25%	99.42%	50.25%	99.42%	0.00%	0.00%
Snail_Mail	69.92%	80.13%	69.47%	79.69%	0.45%	0.44%
SOLFEDGE	98.36%	100.00%	98.36%	100.00%	0.00%	0.00%
Sorter	44.51%	15.09%	44.51%	15.09%	0.00%	0.00%
Space_Attackers	81.50%	81.68%	81.50%	81.68%	0.00%	0.00%
SPIFFS	83.71%	91.70%	83.92%	91.70%	-0.21%	0.00%
Square_Rabbit	83.39%	88.89%	83.21%	88.89%	0.18%	0.00%
stack_vm	75.10%	82.81%	75.10%	82.81%	0.00%	0.00%
Stock_Exchange_Simulator	67.04%	56.44%	67.04%	56.44%	0.00%	0.00%
stream_vm	82.50%	93.16%	77.50%	89.74%	5.00%	3.42%
stream_vm2	80.18%	88.12%	79.73%	88.12%	0.45%	0.00%
Street_map_service	67.12%	59.59%	66.86%	59.14%	0.26%	0.45%
String_Info_Calculator	84.38%	100.00%	84.38%	100.00%	0.00%	0.00%
String_Storage_and_Retrieval	22.11%	33.34%	22.78%	33.85%	-0.67%	-0.51%

Continued on next page

Continued from previous page...

Program Name	Our Fuzzer CCOV		Samples CCOV		Difference	
	Lines ccov	Branches ccov	Lines ccov	Branches ccov	Lines	Branchs
TAINTEDLOVE	61.84%	66.67%	61.84%	66.67%	0.00%	0.00%
Tennis_Ball_Motion_Calculator	93.06%	100.00%	93.06%	100.00%	0.00%	0.00%
Terrible_Ticket_Tracker	31.72%	23.40%	31.72%	23.40%	0.00%	0.00%
TextSearch	88.31%	91.89%	87.38%	90.13%	0.93%	1.76%
TFTTP	74.14%	69.61%	74.14%	69.61%	0.00%	0.00%
The_Longest_Road	93.72%	99.71%	93.72%	99.71%	0.00%	0.00%
Thermal_Controller_v2	21.41%	9.87%	21.41%	9.87%	0.00%	0.00%
Thermal_Controller_v3	13.34%	5.36%	13.34%	5.36%	0.00%	0.00%
TIACA	11.85%	20.83%	11.85%	20.83%	0.00%	0.00%
Tick-A-Tack	90.94%	100.00%	91.26%	100.00%	-0.32%	0.00%
tribute	29.89%	29.17%	29.89%	29.17%	0.00%	0.00%
TVS	84.56%	83.44%	83.27%	83.33%	1.29%	0.11%
university_enrollment	78.44%	94.45%	78.58%	94.73%	-0.14%	-0.28%
User_Manager	81.21%	69.03%	81.21%	69.03%	0.00%	0.00%
UTF-late	85.14%	100.00%	85.53%	100.00%	-0.39%	0.00%
ValveChecks	11.45%	14.64%	11.45%	14.64%	0.00%	0.00%
Vector_Graphics_2	88.44%	94.24%	88.79%	94.45%	-0.35%	-0.21%
Vector_Graphics_Format	76.56%	83.89%	76.49%	83.89%	0.07%	0.00%
Venture_Calculator	90.32%	89.84%	90.32%	89.84%	0.00%	0.00%
vFilter	42.58%	42.22%	41.93%	41.22%	0.65%	1.00%
Virtual_Machine	15.17%	18.00%	15.17%	18.00%	0.00%	0.00%
virtual_pet	91.21%	100.00%	91.21%	100.00%	0.00%	0.00%
Water_Treatment_Facility_Simulator	92.31%	96.93%	92.31%	96.93%	0.00%	0.00%
WhackJack	86.84%	91.29%	87.05%	91.79%	-0.21%	-0.50%
WordCompletion	85.50%	91.67%	85.50%	91.67%	0.00%	0.00%
XStore	3.15%	2.75%	3.15%	2.75%	0.00%	0.00%
yolodex	95.61%	100.00%	96.28%	100.00%	-0.67%	0.00%

6.6 Adjustment Percentages Crashes and Code Coverage

6.6.1 Experiments

The aim the experiment is to test different percentage numbers see Section 6.5.4 to find out which is the best one in the case of number of crashed programs and code coverage. Moreover, the purpose of the testing is to find out the differences between each percentage regrading number of crashes and code coverage. The used percentages are 5%, 7%, 10%, 12% and 15%.

The machine for generating fuzzing files and testing has environment: the CPU is AMD Ryzen Threadripper 2920X 12-Core Processor, memory is 32 GB, Ubuntu 18.04.4 LTS, and OS type 64-bit.

The experiment was to go through all of the programs in the dataset. The testing kept tracking of the number of crashes and the code coverage which was collected for each program. We took the averages for line and branch code coverage for the programs that have multiple files in the source directory. Moreover, the percentage was changed for different percentage numbers: 5%, 7%, 10%, 12% and 15%. These percentages are the percentage number of sample files that can be used to get grammars to generate fuzzing files. Each percentage was running for three rounds and each round took an hour for testing each program. Each round obtained new code coverage information for each program, which means each round testing was not using previously created code coverage. The testing took 6 programs in parallel for testing. This experiment took about 8 weeks to finish testing all programs for all the five percentages.

5% Crashes and Code Coverage

The percentage in the tool adjusted to 5% of number of sample files. After that, the tool was tested for three rounds to get the maximum number of crashes. The maximum number of crashed programs was 71. After each round, the code coverage for each program was collected and stored.

Figure A.1 shows the line code coverage chart for 5%. Figure A.2 shows the branch code coverage chart for 5%.

7% Crashes and Code Coverage

The percentage in the tool changed to 7% of number of sample files. Therefore, the tool was tested for three rounds to get the maximum number of crashes. The number of crashed programs was 71. After each round, the code coverage for each program was collected and stored.

Figure A.3 shows the line code coverage chart for 7%. Figure A.4 shows the branch code coverage chart for 7%.

10% Crashes and Code Coverage

The percentage was changed to 10% of sample files number. After that, the tool was tested for three rounds and has found a maximum number of crashed programs which was 73. After each round, the code coverage for each program was collected and stored.

The line code coverage chart for 10% can be found in Figure A.5. Figure A.6 shows the branch code coverage chart for 10%.

12% Crashes and Code Coverage

The percentage was changed to 12% of sample files number. After that, the tool was tested for three rounds and has found a maximum number of crashed programs which was 71. After each round, the code coverage for each program was collected and stored.

Figure A.7 shows the line code coverage chart for 12%. Figure A.8 shows the branch code coverage chart for 12%.

15% Crashes and Code Coverage

The percentage was changed to 15% of sample files number. After that, the tool was tested for three rounds and has found a maximum number of crashed programs which was 70. After each round, the code coverage for each program was collected and stored.

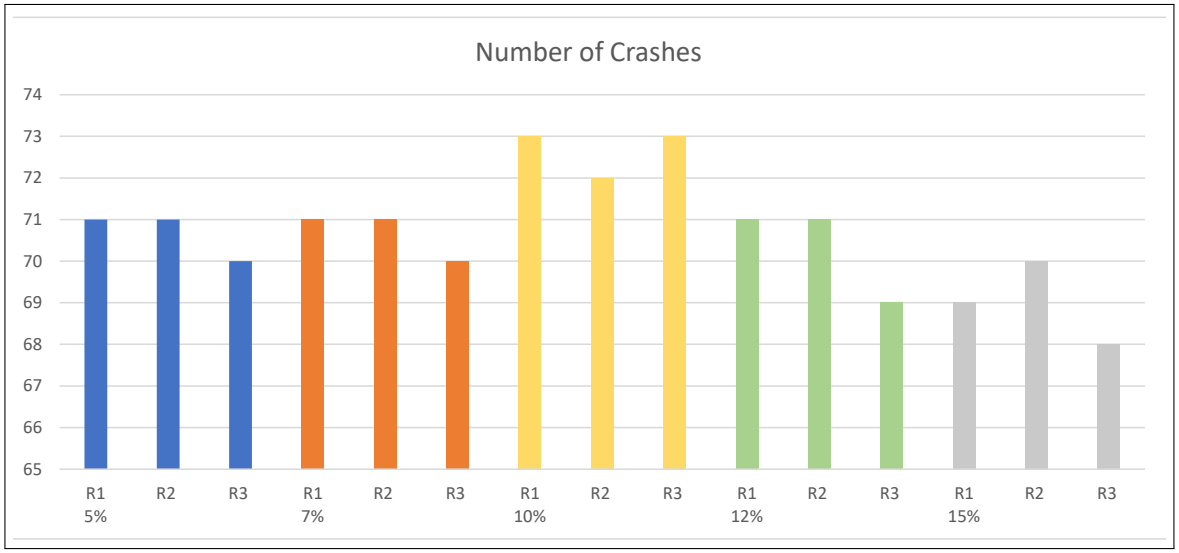
Figure A.9 shows the line code coverage chart for 15%. Figure A.10 shows the branch code coverage chart for 15%.

6.6.2 Percentages Experiments Code Coverage Summary

Number of Crashes

Figure 6.6 shows the summary of number of crashed programs for each percentage in each round.

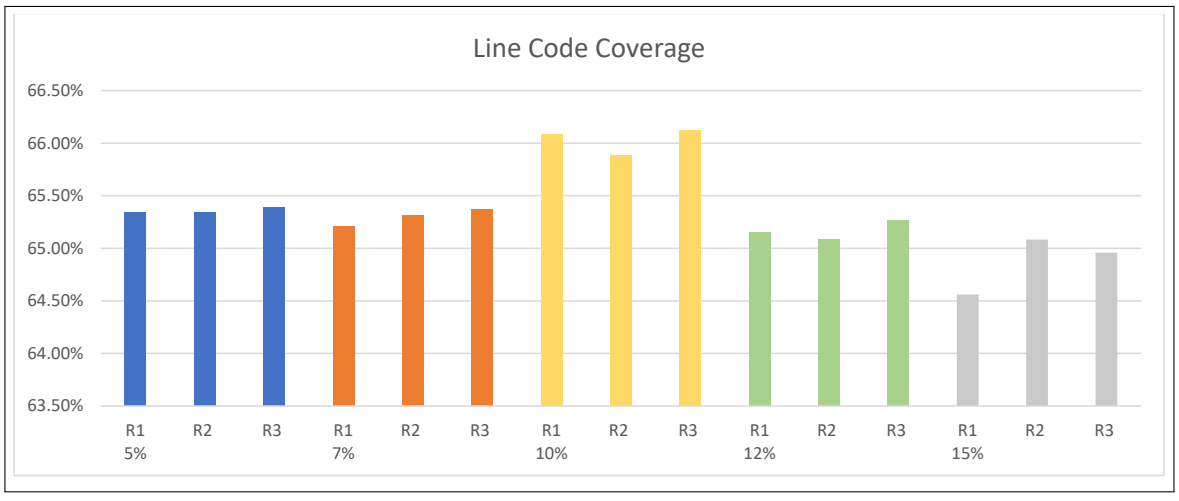
Figure 6.6: Summary of Number of Crashes



Line Code Coverage Summary

Figure 6.7 shows the summary of line code coverage for each percentage in each round.

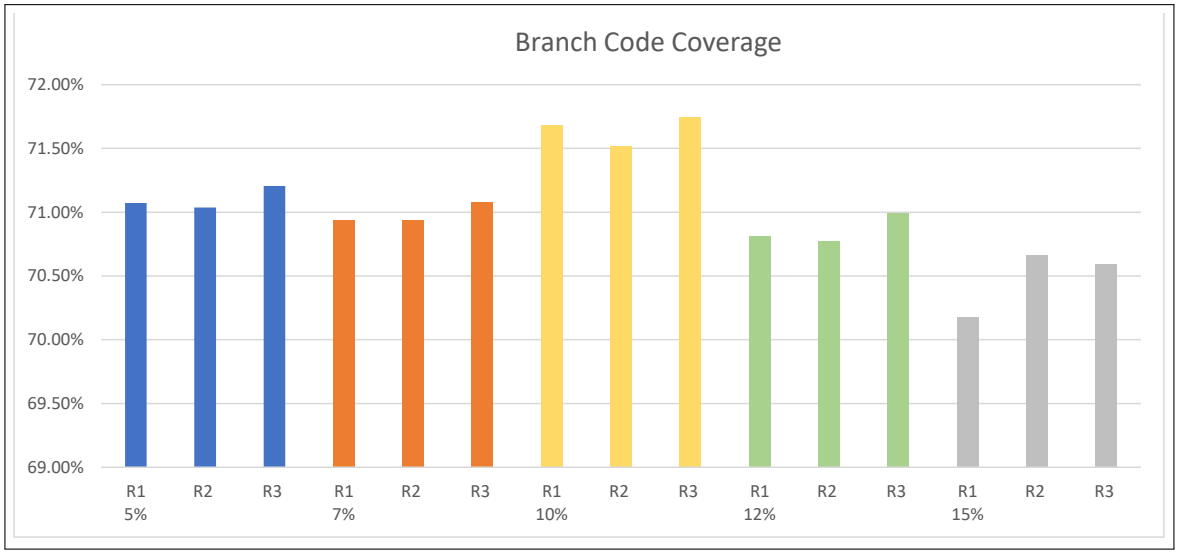
Figure 6.7: Summary of Line Code Coverage



Branch Code Coverage Summary

Figure 6.8 shows the summary of branch code coverage for each percentage in each round.

Figure 6.8: Summary of Line Code Coverage



Lessons Learned

After finishing the experiments about adjusting percentages, we found that the increase in code coverage does not mean the increase of number of crashes. Some programs have high code coverage but did not crash. Also, Some programs have low code coverage but they crashed.

We observed that when the percentage increases from 5% to 10%, the code coverage and crashes increase. Moreover, when the percentage increases after 10%, the code coverage decreases because 5%, 7%, and 10% included the necessary commands and grammars which show increased code coverage and crashes.

We believe percentages 12% and 15% included fewer critical commands and grammars that could lead a program to crash; also, they show decreased code coverage and number

of crashes. We determine the code coverage and crashes decrease when the percentage gets larger than 10%.

We found that 10% is the best percentage for the tool that can lead to increased code coverage and crashes. The 10% is the best to include the important commands and grammars that can help the fuzzer to generate fuzzing files that lead to more crashes and increase the code coverage.

6.7 Experiments on Fuzzer Updated with Markov Chain Model

The experiments had been conducted to examine the fuzzer after using Markov chain model. The experiments begin with testing the programs in DARPA CGC dataset. We used a bash script to loop through each program in the data set. Then, the tool generates test cases to test a program and waits for any crash or exception. During the testing, we look for "core dumped" or "Segmentation fault". In those cases, we know that a bug has been revealed. We conducted our testing on 235 programs in the dataset. For each program, the fuzzer run for one hour. Since fuzzing is very random and the results are usually random, we ran the testing of 235 programs three times.

The machine that had been used for generating fuzzing files and testing has environment: the CPU is AMD Ryzen Threadripper 2920X 12-Core Processor, memory is 32 GiB, Ubuntu 18.04.4 LTS 64-bit, and 2 TB disk space.

6.7.1 Testing Results and Observations

We used the same data set for our previous study, which is CB-multios dataset that has been exported from DARPA CGC DECREE system to Linux system by TrailofBits team [68]. There are 247 programs in CGC DARPA dataset. However, we discarded 12 programs because of compilation and running issues. Therefore, we performed our

investigation and tested on 235 programs.

After completing the testing, we found that 73 programs crashed which is 31.06% of the total programs. Moreover, we found crashes in 7 new programs that were not discovered by our previous study. The fuzzer with Markov chain model should be run on top of our regular fuzzer. If we integrate the fuzzer with Markov chain model into our previous tool, these 7 new crashed programs can be found in addition to the 82 crash set found by our previous tool and would become 89 crashed programs. The list of crashed programs is shown in Table 6.7.

Table 6.7: Crashed Programs by Markov Chain Model

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (m:s)**
ASCII_Content_Server	0:16	1000	5000	1:43
Azurad	0:11	1000	5395	58:41
Bloomy_Sunday	0:3	1000	5058	0:1
CGC_Hangman_Game	0:4	1000	5008	26:2
CGC_Planet_Markup _Language_Parser	0:20	1000	5036	18:9
Charter	0:6	1000	5001	16:46
Checkmate	0:4	1000	5001	1:34
CML	0:5	1000	5092	3:5
DFARS_Sample_Service	0:22	1000	5029	0:5
Diary_Parser	0:5	1000	5006	0:3
Diophantine_Password_Wallet	0:4	1000	5003	1:33
Divelogger2	0:5	1000	5058	0:8
Document_Rendering_Engine	1:34	1000	5148	0:15
Dungeon_Master	0:7	1000	5596	41:13
Eddy	0:11	1000	5000	6:1
electronictrading	0:14	1000	5000	7:11
EternalPass	0:4	1000	5002	0:6
FablesReport	0:14	1000	5000	18:32
Filesystem_Command_Shell	0:33	1000	5033	6:2
Finicky_File_Folder	0:12	1000	5029	0:6

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (m:s)**
FISHYXML	0:4	1000	5001	3:49
Flash_File_System	0:5	1000	5005	2:9
Flight_Routes	0:5	1000	5077	25:11
Game_Night	0:4	1000	5070	3:53
Grit	2:50	1000	5162	46:18
HackMan	0:4	1000	5096	0:5
HighFrequencyTradingAlgo	0:12	1000	5000	0:10
Hug_Game	0:4	1000	5001	2:5
INSULATR	2:44	1000	5026	46:32
KTY_Pretty_Printer	0:4	1000	5062	0:23
Lazybox	0:5	1000	5048	1:2
Loud_Square_Instant _Messaging_Protocol _LSIMP	0:4	1000	5030	24:43
Matchmaker	0:24	1000	5168	1:10
matrices_for_sale	0:4	1000	5007	0:6
Monster_Game	0:3	1000	5004	45:51
Movie_Rental_Service	0:6	1000	5012	1:43
Multicast_Chat_Server	0:45	968	5647	0:54
Multipass3	0:8	1000	5005	49:13
Network_Queuing_Simulator	0:4	1000	5009	0:7
online_job_application	0:4	1000	5022	43:56
OTPSim	0:7	1000	5223	0:48
Pac_for_Edges	0:14	1000	5000	47:26
Palindrome	0:4	1000	5002	0:11
Palindrome2	0:4	1000	5000	0:18
Pipelined	0:4	1000	5006	0:31
pizza_ordering_system	0:3	1000	5106	14:13
Printer	0:4	1000	5065	0:10
PRU	0:5	1000	5003	0:38
PTaaS	0:16	1000	5007	0:11
Query_Calculator	0:5	1000	5139	0:5
Recipe_and_Pantry _Manager	0:5	1000	5018	4:11
Recipe_Database	2:9	1000	5201	19:11
Rejistar	0:17	1000	5000	2:4

Continued on next page

Continued from previous page...

Program name	**Generation Time (m:s)**	**# of Sample input files**	**# of Fuzzing files**	**Time until crash (m:s)**
REMATCH_3–Address_Resolution_Service–SQL_Slammer	0:4	1000	5004	0:23
REMATCH_4–CGCRPC_Server–MS08-067	0:8	1000	5004	51:12
REMATCH_5–File_Explorer–LNK_Bug	0:4	1000	5006	0:28
REMATCH_6–Secure_Server–Heartbleed	0:4	1000	5001	0:2
RRPN	0:10	1000	5000	0:9
Sample_Shipgame	0:4	1000	5065	0:14
SCUBA_Dive_Logging	0:5	1000	5014	4:12
Secure_Compression	0:3	1000	5079	15:1
simple_integer_calculator	0:11	1000	5003	7:21
stream_vm	0:20	1000	5042	0:32
String_Storage_and _Retrieval	0:12	1000	5032	57:11
TAINTEDLOVE	0:7	1000	5000	0:5
Tennis_Ball_Motion _Calculator	0:3	1000	5008	4:41
TVS	0:49	1000	5954	0:49
university_enrollment	0:4	1000	5009	8:17
Vector_Graphics_2	0:7	1000	5001	7:39
Vector_Graphics_Format	0:7	1000	5000	0:5
virtual_pet	0:6	1000	5009	46:12
Water_Treatment _Facility_Simulator	0:4	1000	5057	53:23
WhackJack	0:3	1000	5011	18:53

After finishing the experiments, we believe that Markov chain model helped the tool to try different paths or locations to trigger a possible vulnerability. Moreover, the tool with Markov chain model can get new fuzzing files that can be better than previous tool because with using sample input files to generate fuzzing files can not help the fuzzer to try other paths or locations but in the tool using Markov chain can have different

command order to try fuzzing the target program. The differences between previous tool and the tool using Markov chain is the previous tool uses and modifies sample input files to generate fuzzing files; however, the tool using Markov chain can learn and analyze the sample input files and calculate the probabilities to help the tool generate new fuzzing files. Moreover, the number of generated fuzzing files with the fuzzer using Markov chain is fewer than the number of generated fuzzing files with our previous tool and can find crashed programs faster than previous tool because most of crashed programs can be found in early first or second round but the previous tool can find most crashes after third round.

Chapter 7: Conclusion and Future Work

7.1 Conclusion

Software testing is one of the important steps in software development that makes sure that an application or software is almost free of bugs and vulnerabilities. Fuzzing is one type of software testing. It is an automated technique in testing a program that generates malformed user inputs and inserts them to a program under testing and monitoring for abnormal behavior. Some programmers may overlook or forget about a vulnerability in a software that can cause a software failure.

The research work presented in this dissertation focuses on analyzing sample input files to extract grammars (commands and parameters) and using the extracted grammars to generate fuzzing files. Fuzzing techniques are: 1) modifying the sample input file line by line, and after a line is modified a new fuzzing file is generated; 2) using different kinds of ways, strings (letters, numbers, mix of letters and numbers, etc.) and different lengths to create a fuzzing string; 3) changing the numbers in the input to "[0-9]" to be substituted with random number and length. Then, the tool uses the fuzzing input files to test programs to trigger or reveal bugs. Moreover, Markov chain model was used to learn the commands order and probability to support generating new fuzzing files without modifying the sample input files. It is used to generate completely new fuzzing files in addition to the ones generated by making modifications on the sample input files. These new fuzzing files can be inserted to the programs to discover bugs or vulnerabilities in them.

The main contributions of this dissertation are:

- Reviewing grammar-based fuzzing studies and publishing a paper for it.
- Analyzing sample input files to extract grammars from them.

- Using the extracted grammars from sample input files and generating fuzzing input files.
- Using Markov chain to learn the commands order on top of the tool and generating new fuzzing input files without modifying the sample input files. By learning the order and probability of the commands from sample input files, the new fuzzer can generate fuzzing input files based on that information and the extracted grammar. Also, the new fuzzer goes deeper in the program and thus may reveal more bugs and vulnerabilities.
- Testing and experiments had been conducted and the results are documented in this dissertation.

Our regular tool is able to get grammars form many different types of input file formats such as web browser, photo analyzer, board game, etc. However, other studies in the related work are focused on using a provided grammar for certain types of input formats such as GramFuzz [22] and Learn&Fuzz [54]. Moreover, our tool can learn grammars from sample input files to generate fuzzing input files. However, other studies in the related work are using manually provided grammars to the fuzzing tool such as Skyfire [19]. Moreover, our tool showed that it is faster in triggering vulnerabilities in the tested program than others. By using sample input files during testing, AFL, AFLFast, FairFuzz, and MOPT are not able to find more crashes than our tool because they are not grammar-based and it is difficult for them to get the correct file input structure to generate fuzzing inputs that can help them to trigger the vulnerabilities in different programs. Grammar-based fuzzing supports to get the correct format for a program and uses it to generate fuzzing inputs that leads to possibly crash a program. The issue with our regular tool is when using the sample input file only, it cannot guide the fuzzer to try

other locations or paths. Some sample input files are large, so when the tool generates fuzzing files, it generates larger number of fuzzing files and takes time to test them all.

Our tool using Markov chain model uses the extracted grammars, learns command order and probabilities from sample input files to generate new fuzzing files. The tool can use the same techniques from previous tool to generate fuzzing files. The tool is able to obtain new fuzzing files that can try other locations or paths in a hope of triggering a vulnerability in a target program. The generated fuzzing files can be run in addition to the files generated by our regular fuzzer, so that this is a higher possibility to cause a crash in the program.

The results showed how the performance of our fuzzer. Our regular fuzzer crashed 82 programs. Moreover, we measured the code coverage of our tool, which showed that our tool perform better than sample input files code coverage. Lines code coverage is approximately in average 65% of all tested programs. Also, branch code coverage is about in average of 71%. After we used Markov chain on top of our regular tool, the new fuzzer can crash additional programs. It crashed 7 new programs that were not discovered by our previous tool. If we assume that our new tool was built into the previous one the total crashed will be 89 programs.

7.2 Future Work

While generating effective fuzzing files is still challenging, there is much work to be done. In this dissertation, since the tool used large number of sample input files to generate large amount fuzzing files, the tool should have a way to generate large number of fuzzing files from fewer number of sample input files. In addition, in using Markov chain, there are “START” and “FINAL” mark patterns to indicate the start and the end of commands extracted from sample input files, so if a program has “START” or “FINAL”

as commands, the tool should have other alternatives for these patterns. Moreover, one of the future work could involve using the code coverage feedback information to generate fuzzing inputs for other paths that were not visited. So, the fuzzer can have better code coverage which increase the chance of triggering the vulnerabilities in the target program. Another future work could be testing the fuzzer with other real world applications or dataset to evaluate and improve it. This can be done by looking for other dataset different from CGC DARPA dataset. Moreover, a future work could be running the integration of our tool with Markov chain to get better results. Therefore, after the experiment, the performance of the fuzzer using Markov chain model can be evaluated. Also, testing the fuzzer with Markov chain for 24 hours to see its performance in case of number of crashes. In addition, testing fuzzer with Markov chain model code coverage information and compare it with our previous tool's code coverage. A future work for the fuzzer using Markov chain model is using code coverage information to change the paths that is already found a crash to try other paths to reduce the redundancy. Therefore, the fuzzer can exclude the redundant fuzzing inputs and select other fuzzing inputs that can exercise other locations. Also, the regular fuzzer can employ a machine learning algorithm to use the extracted grammar to generate fuzzing inputs by knowing that each fuzzing input can be different than the others and trying other paths or locations. This can be accomplished by storing the already visited locations and create a queue for generating unique fuzzing inputs. After that, the fuzzer can use them to try and exercise new paths.

Bibliography

[1] M. Dowson, "The Ariane 5 software failure," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 2, p. 84, 1997.

[2] S. Bhattacharya and Y. Nisha, "A case study on boeing's 737 max crisis on account of leadership failure," *International Journal of Research in Engineering, Science and Management*, vol. 3, no. 9, pp. 116–118, 2020.

[3] A. M. Nascimento, L. F. Vismari, P. S. Cugnasca, J. B. C. Júnior, and J. R. de Almeira Júnior, "A cost-sensitive approach to enhance the use of ml classifiers in software testing efforts," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. IEEE, 2019, pp. 1806–1813.

[4] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[6] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[7] C. Paulsen and R. Byers, "NISTIR 7298 Revision 3: Glossary of key information security terms," *[Online]. Available: https://csrc.nist.gov/glossary*, 2019.

[8] R. Shirey, "Internet security glossary," *[Online]. Available: https://www.hjp.at/doc/rfc/rfc4949.html*, 2000.

[9] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, 1998.

[10] MITRE, "CWE: Common weakness enumeration," *[Online]. Available: https://www.cwe.mitre.org/index.html Accessed on: Jan*, 2019.

[11] K. Sneha and G. M. Malle, "Research on software testing techniques and software automation testing tools," in *International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*. IEEE, 2017, pp. 77–81.

[12] B. Laboon, *A Friendly Introduction to Software Testing*. CreateSpace Independent Publishing Platform, 2016.

[13] M. Zalewski, "American fuzzy lop," *[Online]. Available: https://lcamtuf.coredump.cx/afl/*, 2014.

[14] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer. [Online]. Available: https://www.peach.tech/wp-content/uploads/Peach-Fuzzer-Platform-Whitepaper.pdf*, vol. 34, 2011.

[15] K. Serebryany, "Simple guided fuzzing for libraries using llvm's new libfuzzer," 2015.

[16] A. Helin, "Radamsa fuzzer," 2006.

[17] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[19] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Security and Privacy (SP), IEEE Symposium on.* IEEE, 2017, pp. 579–594.

[20] T. Blazytko, M. Bishop, C. Aschermann, J. Cappos, M. Schlögel, N. Korshun, A. Abbasi, M. Schweighauser, S. Schinzel, S. Schumilo *et al.*, "GRIMOIRE: Synthesizing structure while fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1985–2002.

[21] P. Fuzzer, "Discover unknown vulnerabilities," *Peach, Peach Fuzzer.[Online]. Available: http://www. peachfuzzer. com/. Accessed on: Jul*, vol. 13, 2016.

[22] T. Guo, P. Zhang, X. Wang, and Q. Wei, "Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation," in *Informatics and Applications (ICIA), 2013 Second International Conference on.* IEEE, 2013, pp. 212–215.

[23] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security.* Springer, 2016, pp. 581–601.

[24] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[25] Y. Koroglu and F. Wotawa, "Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing," in *IEEE/ACM 14th International Workshop on Automation of Software Test (AST).* IEEE, 2019, pp. 28–34.

[26] P. Amini and A. Portnoy, "Sulley: Pure python fully automated and unattended fuzzing framework," *May*, 2013.

[27] P. Godefroid, "Fuzzing: hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[28] H. Al Salem and J. Song, "A review on grammar-based fuzzing techniques," *International Journal of Computer Science & Security (IJCSS)*, vol. 13, no. 3, p. 114, 2019.

[29] ——, "Using grammar extracted from sample inputs to generate effective fuzzing files," *International Journal of Computer Science & Security (IJCSS)*, vol. 15, no. 5, 2021.

[30] ——, "Grammar-based fuzzing tool using markov chain model to generate new fuzzing inputs," in *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2021.

[31] W. J. Levelt, *An introduction to the theory of formal languages and automata*. John Benjamins Publishing, 2008.

[32] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[33] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Traon, "Zest: Validity fuzzing and parametric generators for effective random testing," *arXiv preprint arXiv:1812.00078*, 2018.

[34] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[35] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[36] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," vol. 51, no. 3, May 2018.

[37] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, "The mayhem cyber reasoning system," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 52–60, 2018.

[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[39] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[40] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[41] J. Newsome and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *In Proceedings of the 12th Network and Distributed Systems Security Symposium*. Citeseer, 2005.

[42] G. Liang, L. Liao, X. Xu, J. Du, G. Li, and H. Zhao, "Effective fuzzing based on dynamic taint analysis," in *2013 Ninth International Conference on Computational Intelligence and Security*. IEEE, 2013, pp. 615–619.

[43] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 196–206.

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

[45] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 474–484.

[46] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 497–512.

[47] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 49–64.

[48] J. Ruderman, "Introducing Jsfunfuzz," *[Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz*, 2007.

[49] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *USENIX Security Symposium*, 2012, pp. 445–458.

[50] S. Y. Kim, S. Cha, and D.-H. Bae, "Automatic and lightweight grammar generation for fuzz testing," *Computers & Security*, vol. 36, pp. 1–11, 2013.

[51] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan, "Grammar-based fuzzing," in *2018 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2018, pp. 32–35.

[52] D. Yang, Y. Zhang, and Q. Liu, "Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012, pp. 1070–1076.

[53] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: an automatic random fuzzer for common file formats," in *Proceedings of the 9th International Symposium on Haskell*. ACM, 2016, pp. 13–20.

[54] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[55] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: a gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 2018, pp. 138–145.

[56] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 2018, pp. 45–48.

[57] P. M. Maurer, "Generating test data with enhanced context-free grammars," *Ieee Software*, vol. 7, no. 4, pp. 50–55, 1990.

[58] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[59] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," *arXiv preprint arXiv:1812.01197*, 2018.

[60] J. Norris, *Markov Chain (Cambridge Series in Statistical and Probabilistic Mathematics)*. Cambridge, U.K.: Cambridge Univ. Press, 07 1998.

[61] J. Cao, X. Liu, H. Guo, L. Cai, and Y. Hu, "Test case generation for web application based on markov reward process," in *Journal of Physics: Conference Series*, vol. 1792, no. 1. IOP Publishing, 2021, p. 012039.

[62] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[63] Y. Ouyang, Z. Wu, Q. Mu, and Q. Wang, "Leveraging markov chain and optimal mutation strategy for smart fuzzing," 2014.

[64] Y. Wang, F. Ye, X. Zhu, and C. Wu, "A method for software reliability test case design based on markov chain usage model," in *2013 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE)*. IEEE, 2013, pp. 1207–1210.

[65] K. Zhou, X. Wang, G. Hou, J. Wang, and S. Ai, "Software reliability test based on markov usage model." *JSW*, vol. 7, no. 9, pp. 2061–2068, 2012.

[66] S. J. Prowell, "Using markov chain usage models to test complex systems," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences.* IEEE, 2005, pp. 318c–318c.

[67] L. Ferres, "Memory management in C: The heap and the stack," *Department of Computer Science, Universidad de Concepcion*, 2010.

[68] Trailofbits, "Darpa challenge binaries on multiple os systems," *[Online]. Available: https://github.com/trailofbits/cb-multios/ Accessed on: March*, 2019.

[69] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

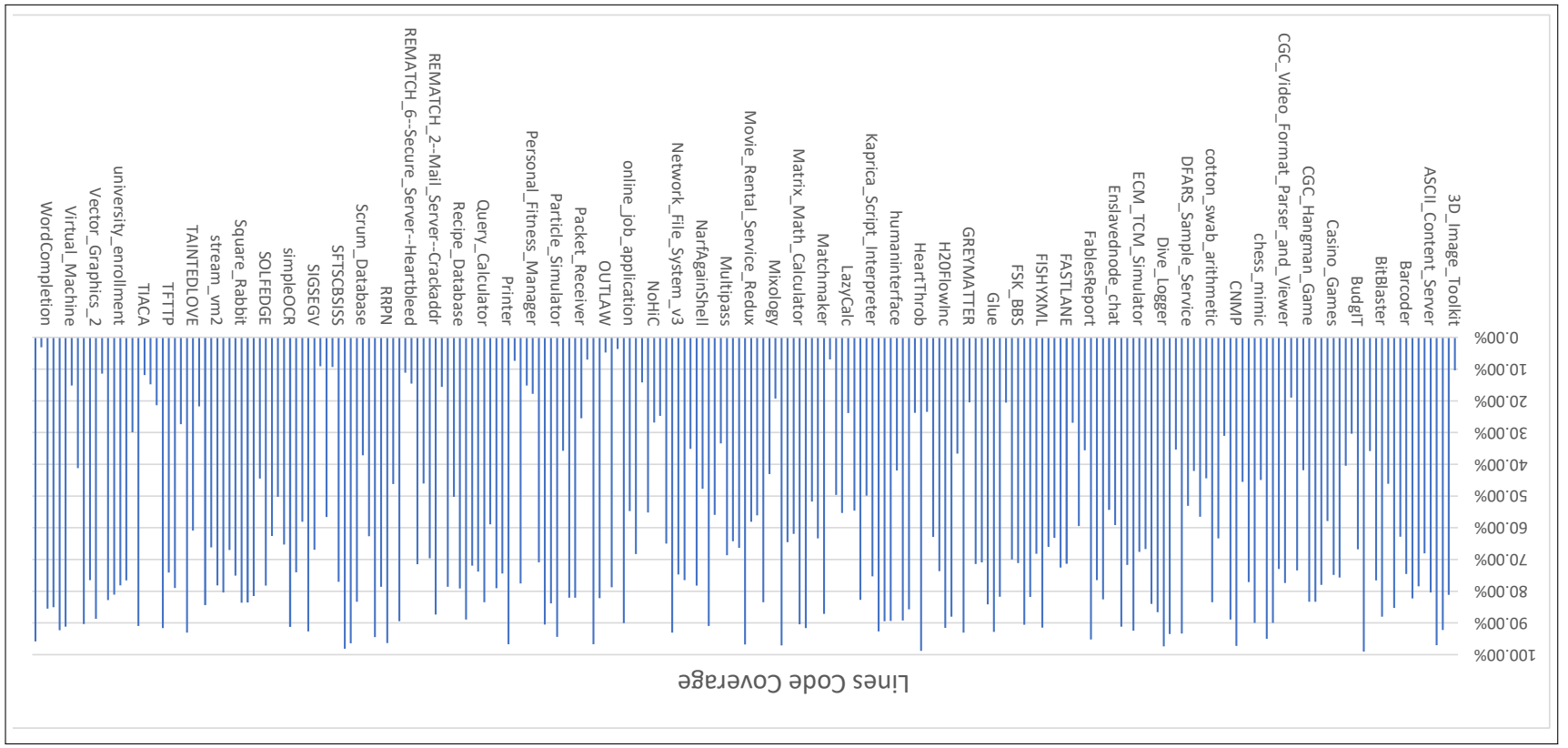
[70] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[71] B. Lou and J. Song, "A study on using code coverage information extracted from binary to guide fuzzing," *International Journal of Computer Science and Security (IJCSS)*, vol. 14, no. 5, 2020.

[72] LLVM_Project, "llvm-cov - emit coverage information," *[Online]. Available: https://llvm.org/docs/CommandGuide/llvm-cov.html Accessed on: Feb*, 2021.

Appendix A: Appendix

Figure A.1: Line Coverage of the Programs after Adjusting the Percentage to 5%.



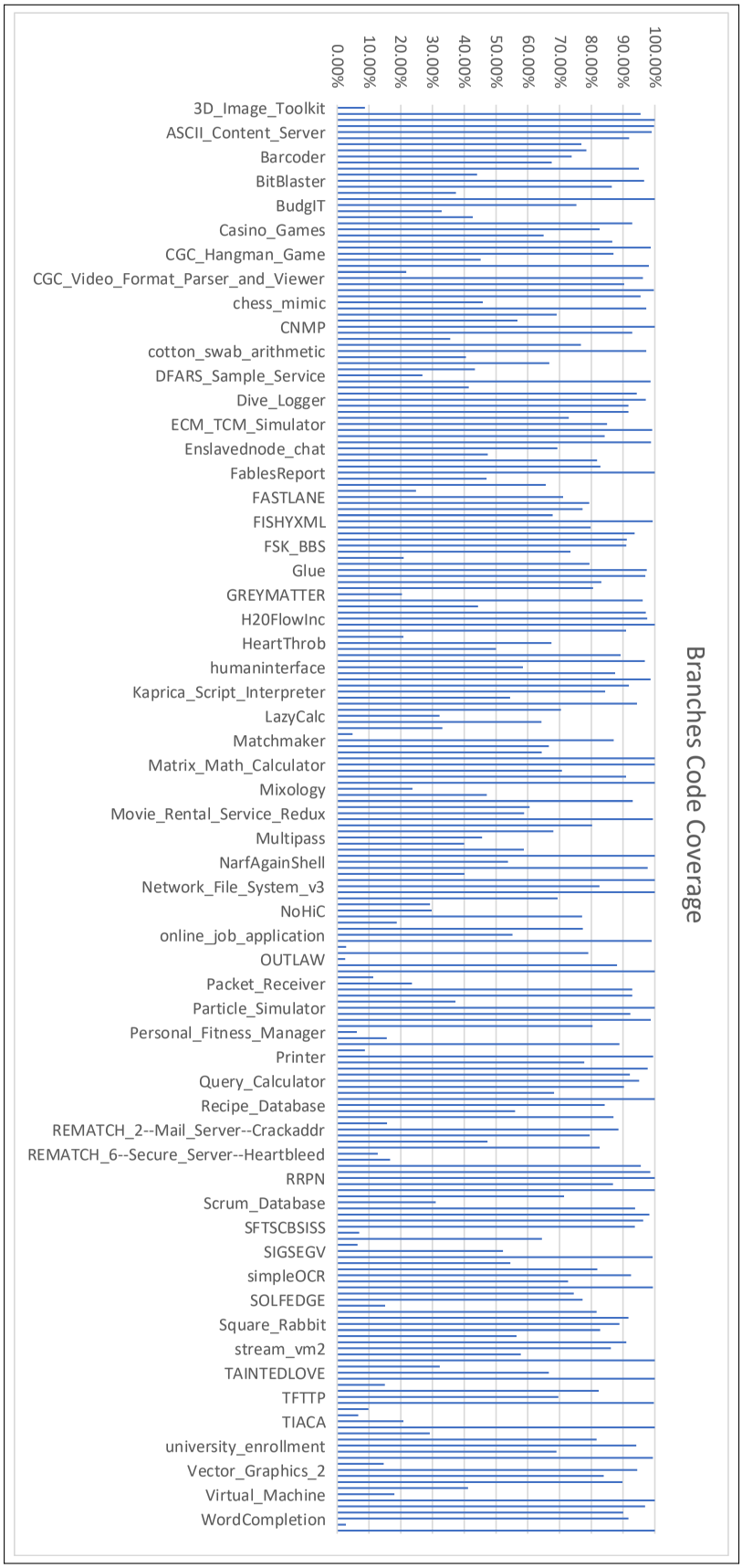


Figure A.2: Branch Coverage of the Programs after Adjusting the Percentage to 5%.

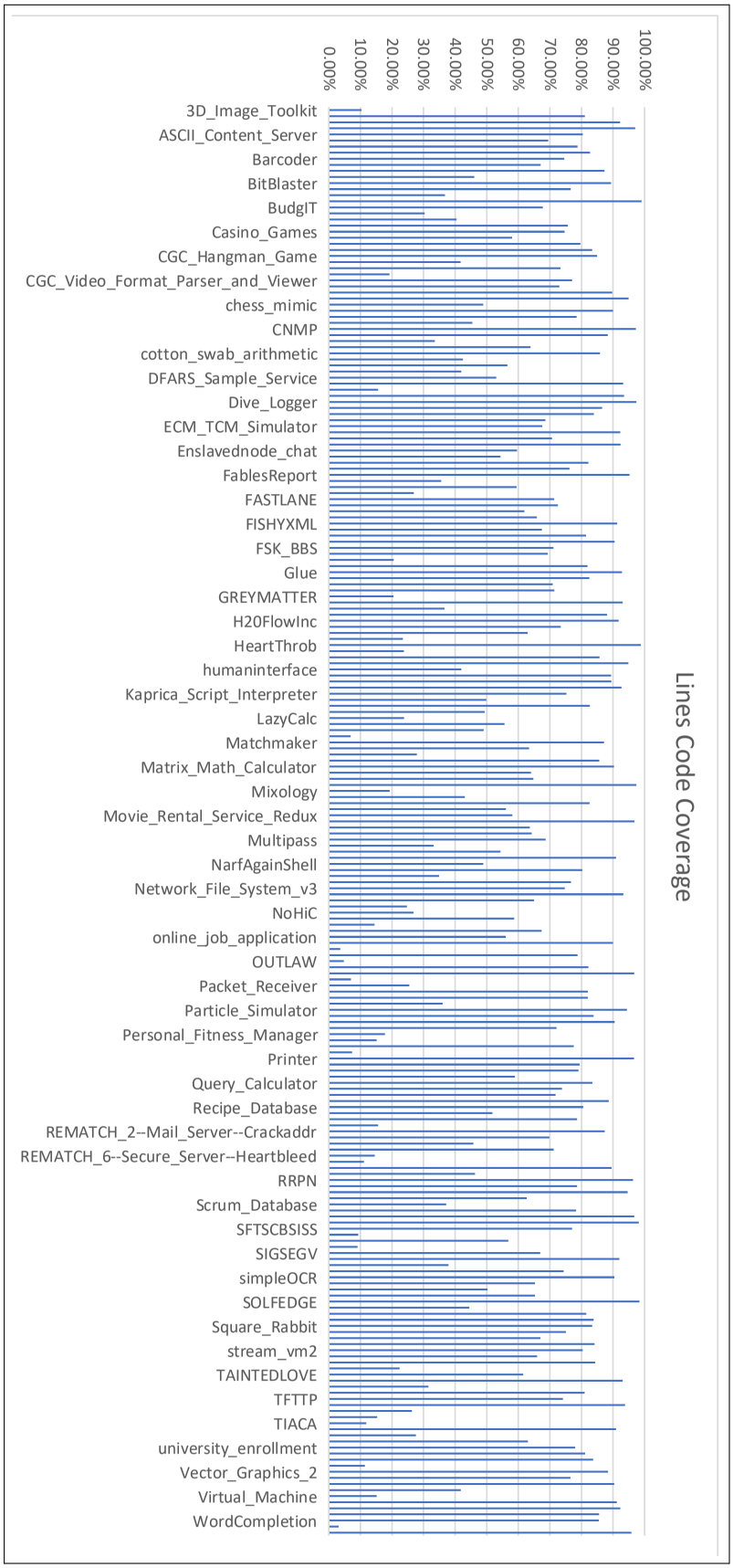


Figure A.3: Line Coverage of the Programs after Adjusting the Percentage to 7%.

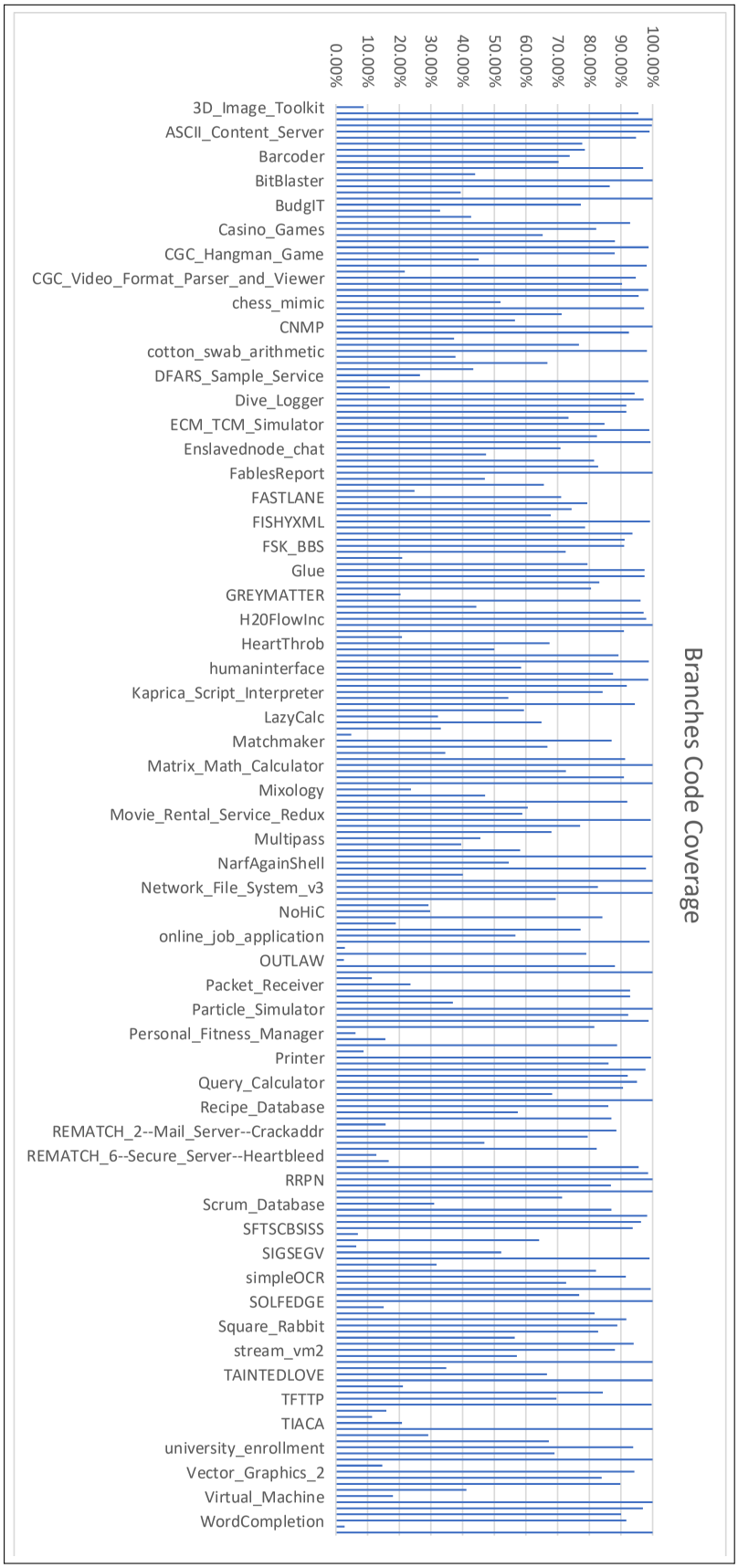


Figure A.4: Branch Coverage of the Programs after Adjusting the Percentage to 7%.

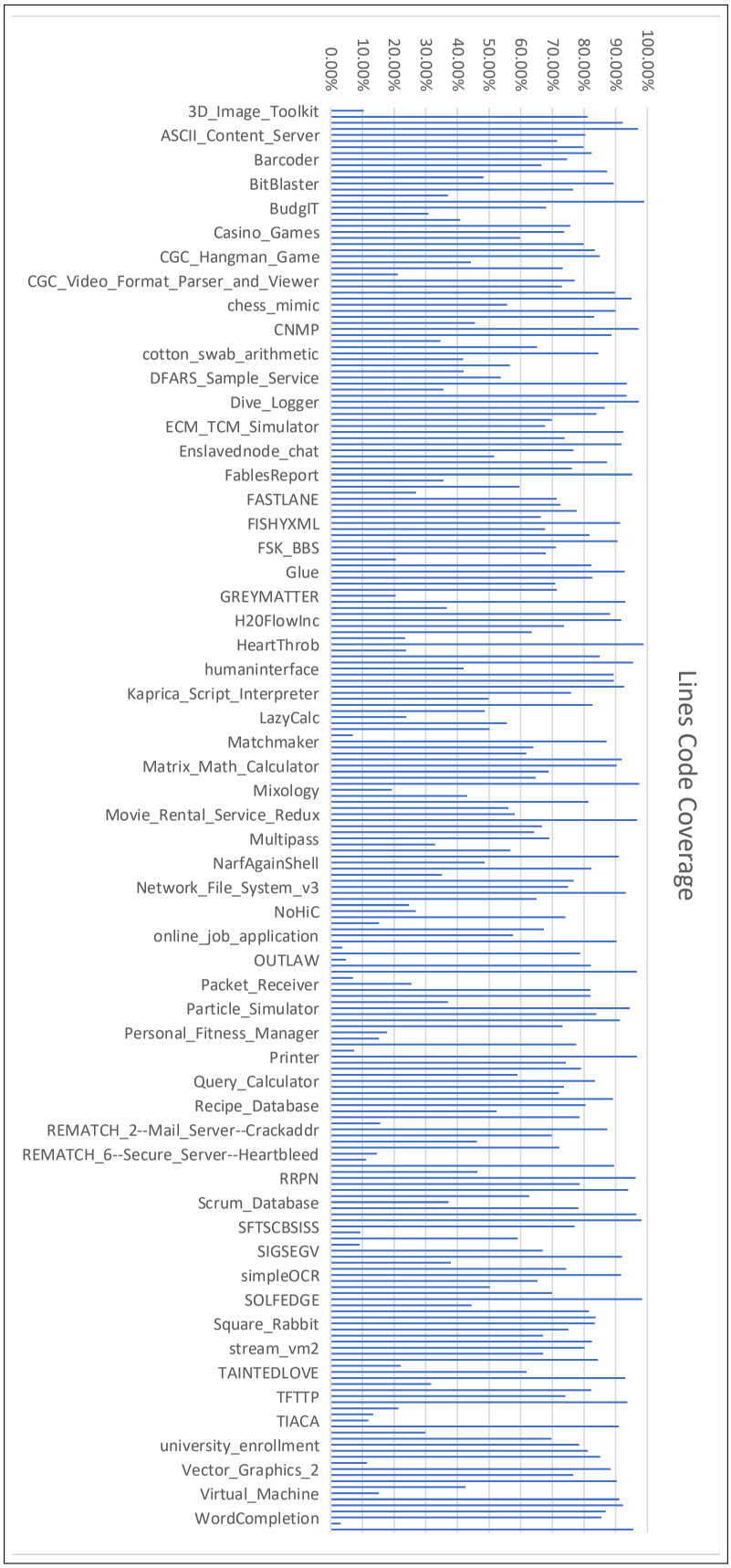


Figure A.5: Line Coverage of the Programs after Adjusting the Percentage to 10%.

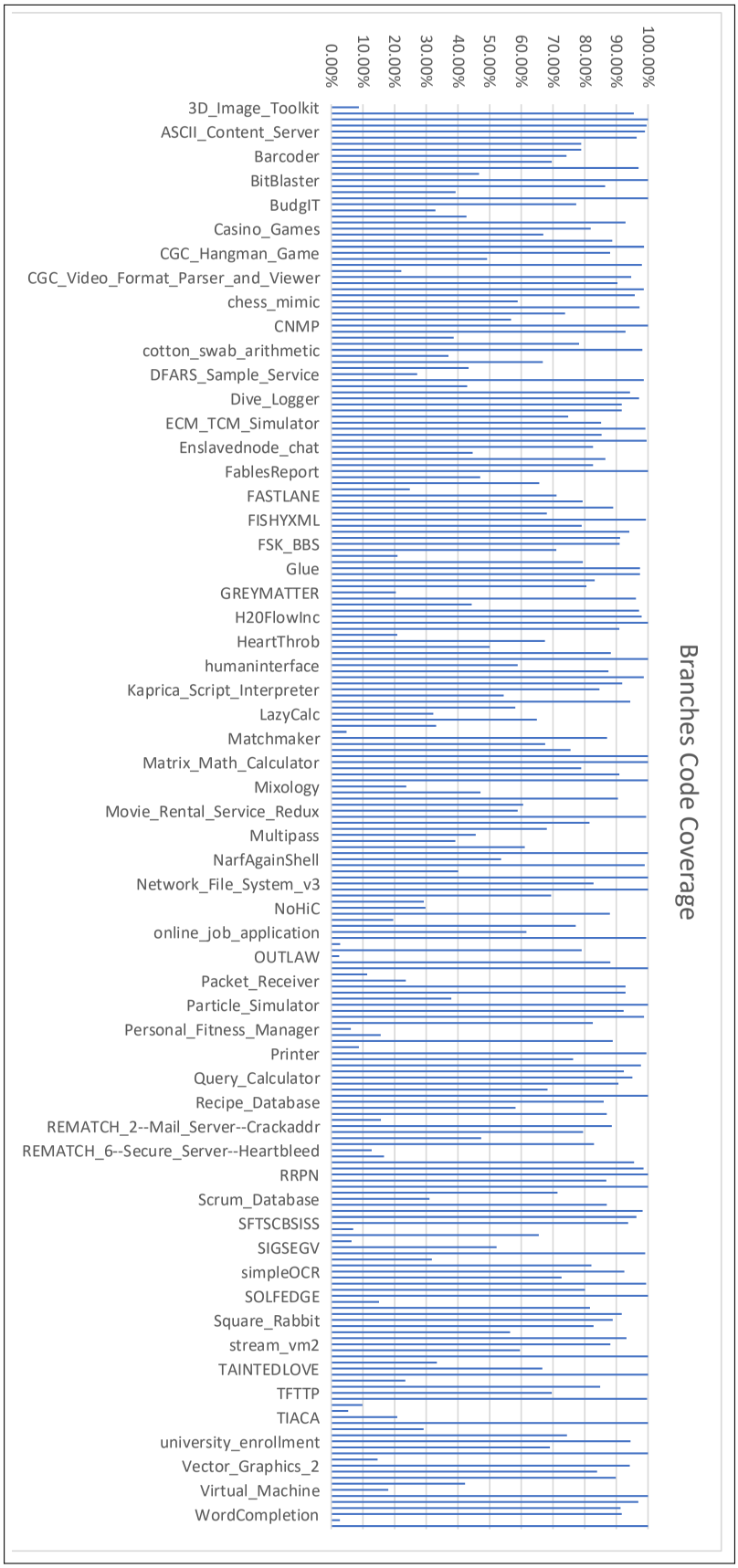


Figure A.6: Branch Coverage of the Programs after Adjusting the Percentage to 10%.

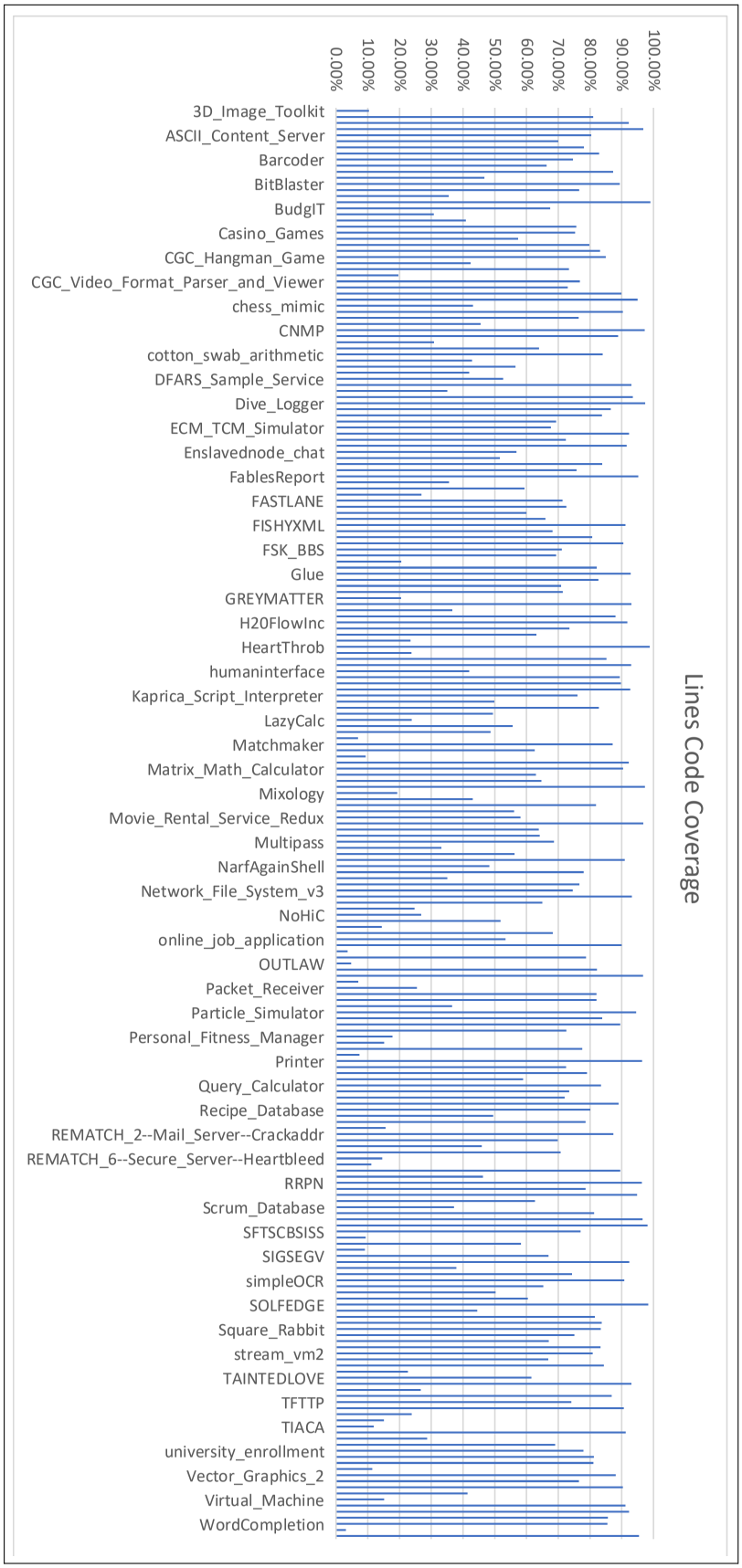


Figure A.7: Line Coverage of the Programs after Adjusting the Percentage to 12%.

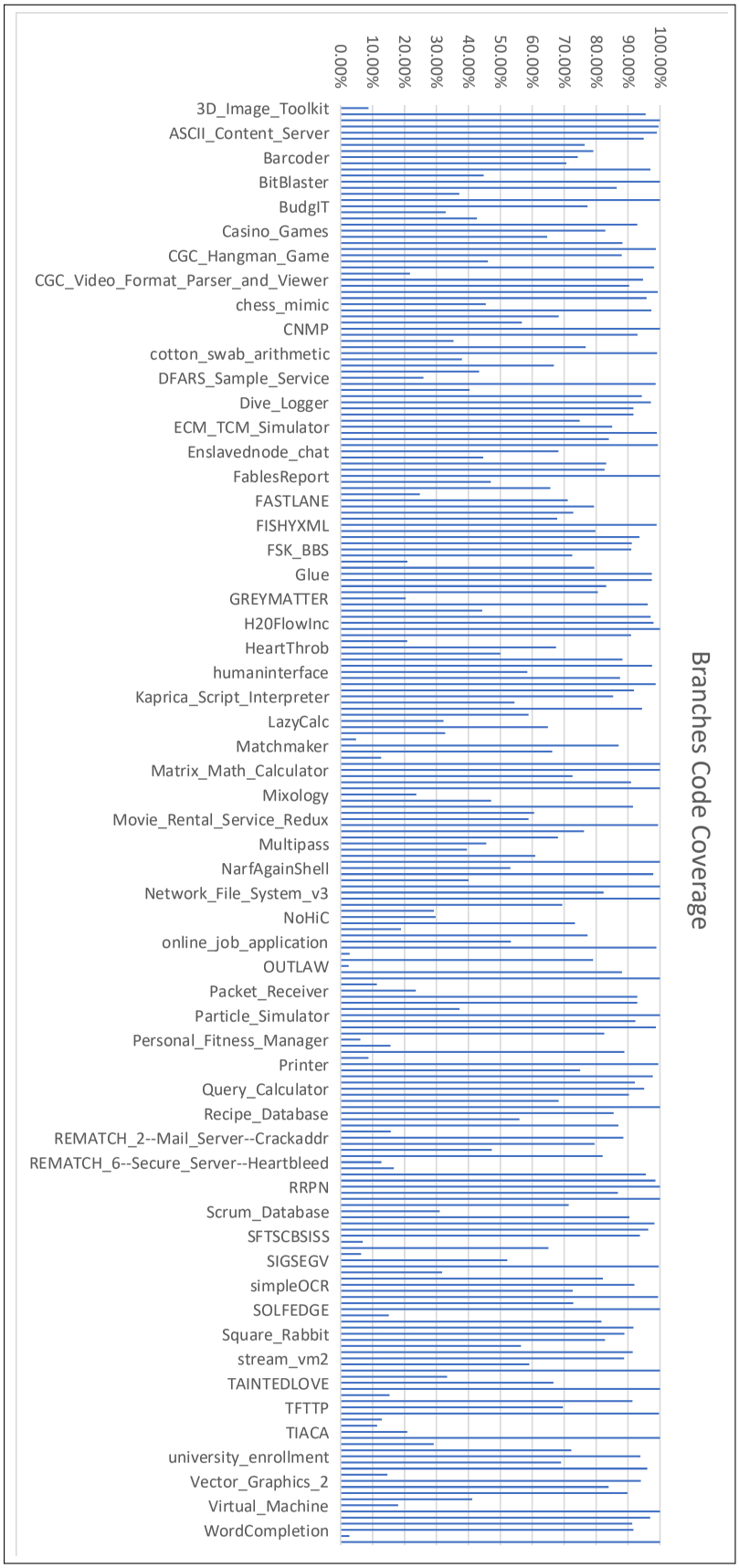


Figure A.8: Branch Coverage of the Programs after Adjusting the Percentage to 12%.

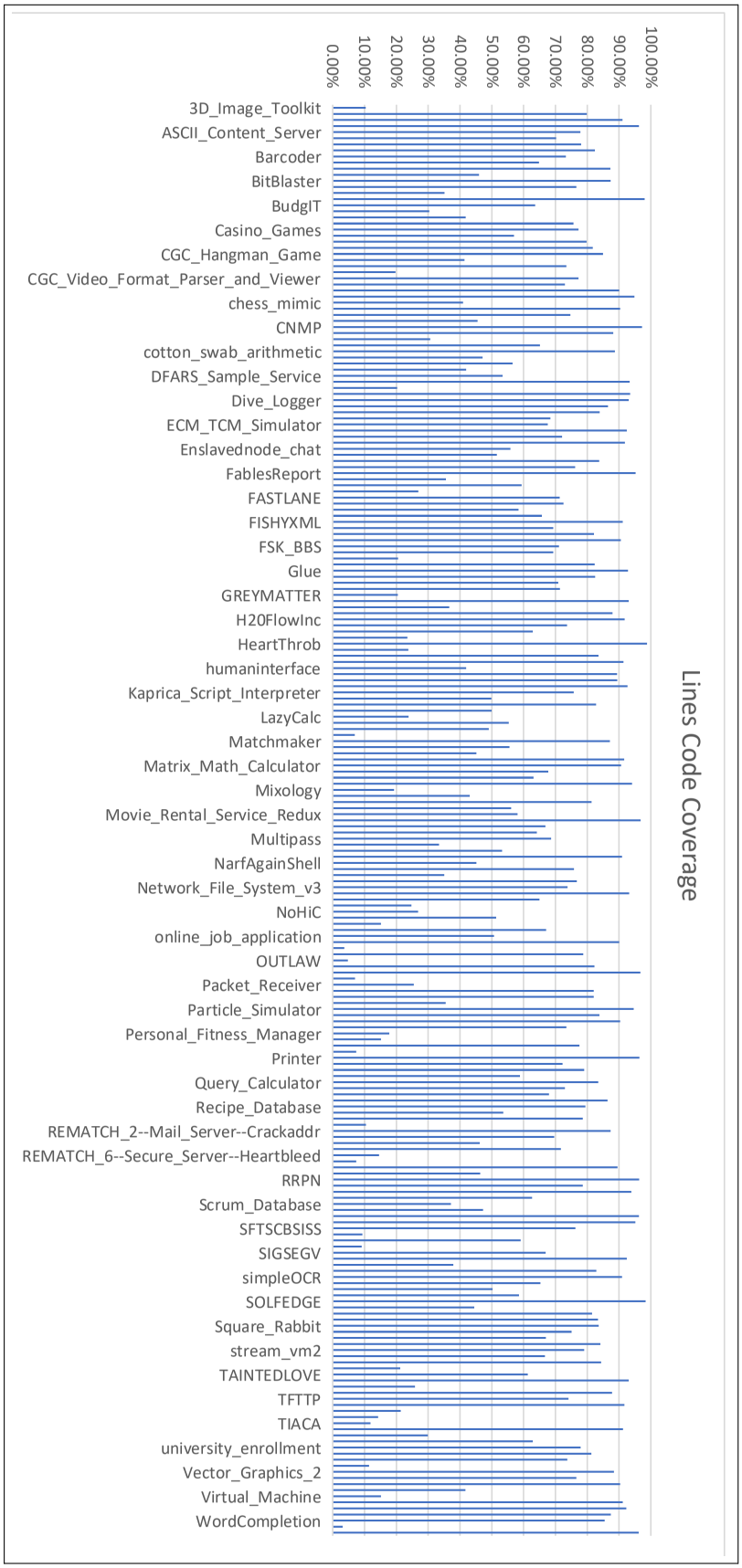


Figure A.9: Line Coverage of the Programs after Adjusting the Percentage to 15%.

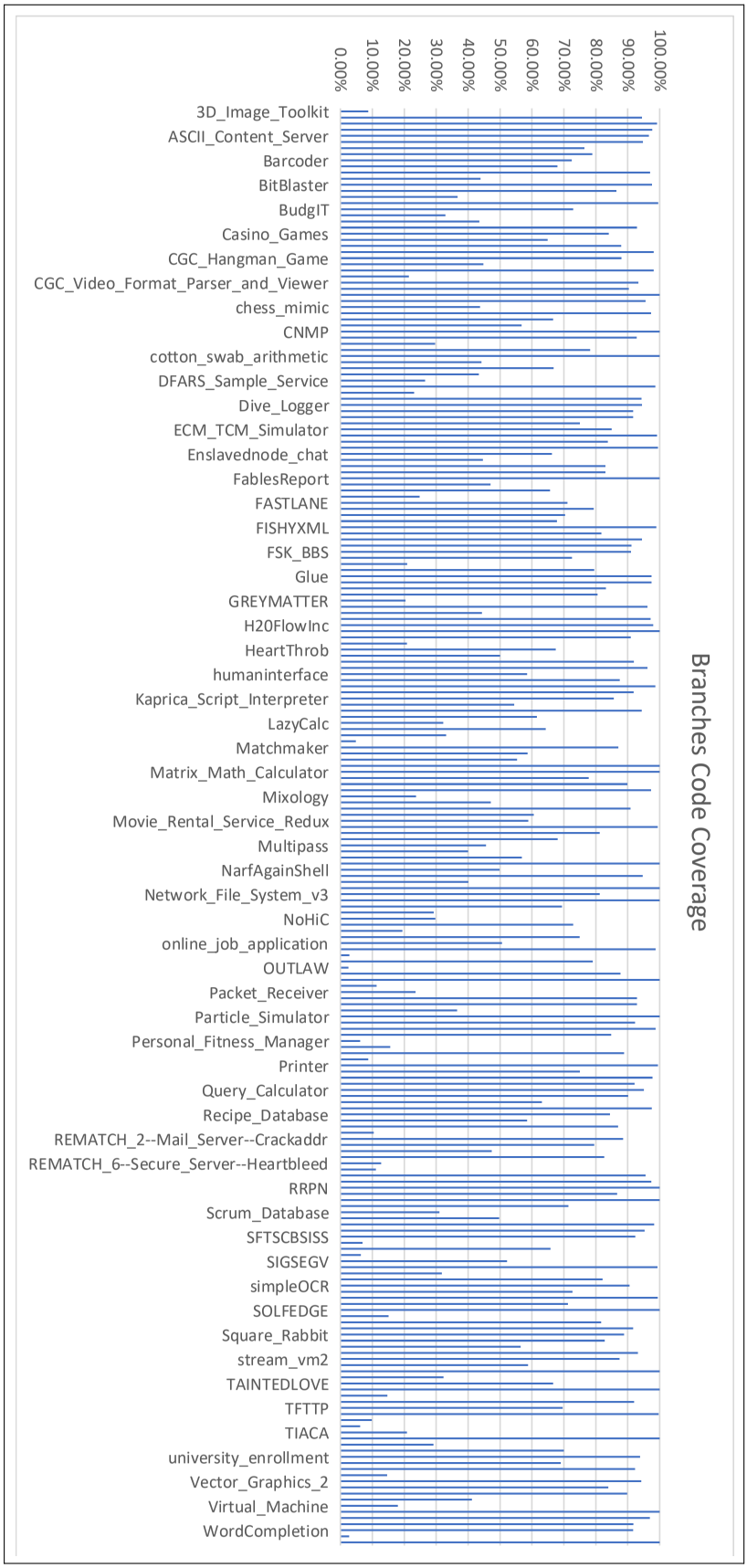


Figure A.10: Branch Coverage of the Programs after Adjusting the Percentage to 15%.