

INTERNATIONAL JOURNAL OF
COMPUTER SCIENCE AND SECURITY (IJCSS)

ISSN : 1985-1553

Publication Frequency: 6 Issues / Year



CSC PUBLISHERS
<http://www.cscjournals.org>

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND SECURITY (IJCSS)

VOLUME 15, ISSUE 5, 2021

**EDITED BY
DR. NABEEL TAHIR**

ISSN (Online): 1985-1553

International Journal of Computer Science and Security is published both in traditional paper form and in Internet. This journal is published at the website <https://www.cscjournals.org>, maintained by Computer Science Journals (CSC Journals), Malaysia.

IJCSS Journal is a part of CSC Publishers

Computer Science Journals

<https://www.cscjournals.org>

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND SECURITY (IJCSS)

Book: Volume 15, Issue 5, October 2021

Publishing Date: 31-10-2021

ISSN (Online): 1985 -1553

This work is subjected to copyright. All rights are reserved whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication of parts thereof is permitted only under the provision of the copyright law 1965, in its current version, and permission of use must always be obtained from CSC Publishers.

IJCSS Journal is a part of CSC Publishers

<https://www.cscjournals.org>

© IJCSS Journal

Published in Malaysia

Typesetting: Camera-ready by author, data conversion by CSC Publishing Services – CSC Journals, Malaysia

CSC Publishers, 2021

EDITORIAL BOARD

EDITOR-IN-CHIEF (EiC)

Professor Chen-Chi Shing
Radford University
United States of America

EDITORIAL BOARD MEMBERS (EBMs)

Professor Ren-Junn Hwang
Tamkang University
Taiwan

Dr. Riccardo Colella
University of Salento
Italy

Dr. Alfonso Rodriguez
University of Bio-Bio
Chile

Professor Abdel-Badeeh M. Salem
Ain Shams University
Egyptian

Associate Professor Fahri Özsungur
Mersin University
Turkey

Dr. Sourav Dutta
Huawei Ireland Research Centre
Ireland

Dr. Sherif E. Abdelhamid
Arab Academy for Science and Technology / Virginia Polytechnic Institute and State University
United States of America

Mr. Rajasekhar Chaganti
Expediagroup Inc
United States of America

Dr. Miaomiao Zhang
Manhattan College
United States of America

Dr. Anna Formica
National Research Council
Italy

Dr. Sanaa Kaddoura

Department of Computing and Applied Technology, Zayed University
United Arab Emirates

Dr. Francesco Taglino

National Research Council
Italy

Dr. Rowanda Ahmed

Uskudar University
Turkey

TABLE OF CONTENTS

Volume 15, Issue 5, October 2021

Pages

- | | |
|-----------|---|
| 146 – 168 | Using Grammar Extracted from Sample Inputs to Generate Effective Fuzzing Files
<i>Hamad Al Salem, Jia Song</i> |
| 169 – 180 | MVC Architecture from Maintenance Quality Attributes Perspective
<i>Safia Nahhas</i> |

EDITORIAL PREFACE

This is *Fifth Issue* of Volume *Fifteen* of the International Journal of Computer Science and Security (IJCSS). IJCSS is an International refereed journal for publication of current research in computer science and computer security technologies. IJCSS publishes research papers dealing primarily with the technological aspects of computer science in general and computer security in particular. Publications of IJCSS are beneficial for researchers, academics, scholars, advanced students, practitioners, and those seeking an update on current experience, state of the art research theories and future prospects in relation to computer science in general but specific to computer security studies. Some important topics cover by IJCSS are databases, electronic commerce, multimedia, bioinformatics, signal processing, image processing, access control, computer security, cryptography, communications and data security, etc.

The initial efforts helped to shape the editorial policy and to sharpen the focus of the journal. Started with Volume 15, 2021, IJCSS appears with more focused issues. Besides normal publications, IJCSS intend to organized special issues on more focused topics. Each special issue will have a designated editor (editors) – either member of the editorial board or another recognized specialist in the respective field.

This journal publishes new dissertations and state of the art research to target its readership that not only includes researchers, industrialists and scientist but also advanced students and practitioners. The aim of IJCSS is to publish research which is not only technically proficient, but contains innovation or information for our international readers. In order to position IJCSS as one of the top International journal in computer science and security, a group of highly valuable and senior International scholars are serving its Editorial Board who ensures that each issue must publish qualitative research articles from International research communities relevant to Computer science and security fields.

IJCSS editors understand that how much it is important for authors and researchers to have their work published with a minimum delay after submission of their papers. They also strongly believe that the direct communication between the editors and authors are important for the welfare, quality and wellbeing of the Journal and its readers. Therefore, all activities from paper submission to paper publication are controlled through electronic systems that include electronic submission, editorial panel and review system that ensures rapid decision with least delays in the publication processes.

To build its international reputation, we are disseminating the publication information through Google Scholar, J-Gate, Docstoc, Scribd, Slideshare, Bibsonomy and many more. Our International Editors are working on establishing good abstracting and indexing listing and a good impact factor for IJCSS. We would like to remind you that the success of our journal depends directly on the number of quality articles submitted for review. Accordingly, we would like to request your participation by submitting quality manuscripts for review and encouraging your colleagues to submit quality manuscripts for review. One of the great benefits we can provide to our prospective authors is the mentoring nature of our review process. IJCSS provides authors with high quality, helpful reviews that are shaped to assist authors in improving their manuscripts.

Editorial Board Members

International Journal of Computer Science and Security (IJCSS)

Using Grammar Extracted from Sample Inputs to Generate Effective Fuzzing Files

Hamad Al Salem

*Computer Science Department
University of Idaho
Moscow, ID, 83844, USA
Najran University, Najran, SA*

alsa5294@vandals.uidaho.edu

Jia Song

*Computer Science Department
University of Idaho
Moscow, ID, 83844, USA*

jsong@uidaho.edu

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 one of the software testing techniques which feeds random input to programs and monitors for abnormal behaviors such as a program crash. One of the limitations of fuzzing is that most of the fuzzers require highly structured input or certain input pattern; otherwise, the fuzz testing may be terminated at the early stage of the program execution because of not meeting the input format requirements. Some fuzzers resolve this problem by manually creating program specific input grammars to help guide fuzzing, which is tedious, error prone, and time consuming. In addition, 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 paper. 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 grammar from them automatically and hence generate program specific fuzzing files. The fuzzing tool is fast and can find a crash in a short time. From the experiments, it successfully crashed 79 (out of 235) programs of the DARPA CGC dataset.

Keywords: Software Testing, Fuzzing, Grammar Analysis, Security Testing.

1. INTRODUCTION

Recently, software security testing has been widely used to reduce the number of bugs in the software. Researchers have proposed different approaches to support automated software security testing. Fuzzing is one of the approaches. Fuzzing was first attempted by Barton Miller in 1988 to fuzz Unix utilities (Miller, Fredriksen, & So, 1990). Since that time, fuzzing became an interesting topic.

Fuzzing is an automated technique that supports discovering vulnerabilities and weaknesses in a target program by using random generated malformed input data (Oehlert, 2005). 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. 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 (Liang, Pei, Jia, Shen, & Zhang, Sept. 2018). First, an important component in fuzzing process is user input, which is the route in discovery of bugs or flaws in a software/system. Second, the target program

is the software under test that is needed to conduct fuzzing process. Third, fuzzing technique is the strategy of creating fuzzing inputs, which is a part of fuzzing process which creates malformed inputs that could trigger vulnerabilities in the target program. In the end, an abnormal behavior monitors for any exception after running the target program with the malformed fuzzing inputs. Therefore, 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 (Liang, Pei, Jia, Shen, & Zhang, Sept. 2018).

In recent years, fuzzing has been an interesting topic in software testing. There are many fuzzing tools that automatically generate malformed test inputs and feed them to the target program for exploring any bugs; the most known fuzzing tools are AFL (Zalewski, 2014), Peach (Fuzzer, 2016), LibFuzzer (Serebryany, 2015), and RADAMSA (Helin, 2006). 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 (Godefroid, Fuzzing: Hack, art, and science, Jan, 2020). Coverage-guided fuzzing uses trivial program coverage feedback to follow how the running path of the program switches via given fuzzed input (Blazytko, et al., 2019). The fuzzer employs the collected instrumentation data to choose which inputs could be ignored or kept in the corpus queue (Blazytko, et al., 2019). 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 (Noller, Sept. 2018). Dynamic taint analysis is a type of data flow analysis method which is used in many domains like software engineering, and computer security (Gan, et al., Aug. 2020). It can be used in fuzzing to track the generation of some kind of inputs to collect useful data to fuzz programs with different inputs (Gan, et al., Aug. 2020). 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 (Al Salem & Song, June 2019). A detailed literature review on the existing grammar-based fuzzers can be found in Section 3.

2. MOTIVATION AND CONTRIBUTIONS

2.1 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 of a software; however, most of the randomly generated inputs are rejected in the early stage by the target software without visiting interesting locations in the target code. There are many studies conducted on this approach to explore new effective ways to generate interesting fuzzing inputs which are able to trigger a vulnerability deeply in a program (Wang, Chen , Wei, & Liu, 2017). Although great progress has been accomplished in fuzzing, there is still human intervention in the fuzzing process (Blazytko, et al., 2019). It is an important goal for fuzzing 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 it, some fuzzers require valid sample input to at least have a good start point, but they cannot guarantee that a generated fuzzing input meets 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 (Wang, Chen , Wei, & Liu, 2017), (Fuzzer, 2016), (Guo, Zhang, Wang, & Wei, 2013), (Veggalam, Rawat, Haller, & Bos, 2016). For example, fuzzer GramFuzz (Guo, Zhang, Wang, & Wei, 2013) 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 (Aschermann, et al., 2019), (Koroglu & Wotawa, 2019), (Fuzzer, 2016), (Amini, 2013), 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 focused on user inputs which studied how to generate effective fuzzing inputs (Godefroid, Jan, 2020). Others concentrated on fuzzing techniques that exercised fewer resources and reduced the space of potential inputs (Aschermann, et al., 2019). The scope of this research is about learning grammar from the sample input file and generating effective fuzzing inputs from grammars. In the 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.

2.2 Contributions

Most previous research studies centered on generating fuzzing files on a specific input language and use public grammar rules to generate fuzzing input files. The proposed tool can analyze sample input files, obtain the grammar rules, and use them to generate fuzzing inputs automatically. The proposed 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 proposed tool will be designed and implemented based on grammars and effective fuzzing techniques that will improve the efficiency of vulnerabilities detection. without any human interaction, by using input files, the tool can automatically learn the grammar and generate effective fuzzing inputs which can 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 which can explore bugs deeply in the target program. Therefore, the tool will support software developers and industry to find bugs in a system. Since the tool is automated, it will save time and energy of the developers and testers.

3. RELATED WORK

Grammar-based fuzzing is a fuzzing technique that takes a particular input format to get the correct grammars structure. Then, 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 and problems 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 (Kim, Cha, & Bae, 2013). Most of the techniques that use grammar-based fuzzing are utilizing grammar 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 has an efficient way to get fuzzing user input that supports finding deep bugs while using it with

grammar-guided fuzzing (Guo, Zhang, Wang, & Wei, 2013). 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. 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.

3.1 Grammar-based Fuzzers based on Mutation

Some studies used mutation technique in a grammar-based fuzzing such as GramFuzz (Guo, Zhang, Wang, & Wei, 2013), SD-Gen (Sargsyan, et al., 2018), BlendFuzz (Yang, Zhang, & Liu, 2012), QuickFuzz (Grieco, Ceresa, & Buiras, 2016), LangFuzz (Holler, Herzig, & Zeller, 2012), and mutated grammar fuzzer (MGF) (Koroglu & Wotawa, 2019). When a fuzzing tool combines grammars with mutation the generated test cases will be more effective (Guo, Zhang, Wang, & Wei, 2013). Mutation has different ways to mutate and alter the user input. GramFuzz (Guo, Zhang, Wang, & Wei, 2013), BlendFuzz (Yang, Zhang, & Liu, 2012), SD-Gen (Sargsyan, et al., 2018), and LangFuzz (Holler, Herzig, & Zeller, 2012) are similar that after extracting the grammars from user/sample inputs, they mutate some parts of the user inputs based on the user input type to get fuzzing input that may find a vulnerability in the tested program. QuickFuzz (Grieco, Ceresa, & Buiras, 2016) combines bit-level fuzzers with the fuzzing tool to support mutate user input in certain locations. However, Koroglu et al. (2019) stated that a grammar is provided to the fuzzer; then the tool mutated it to generate the fuzzing inputs.

Fuzzing tools GramFuzz (Guo, Zhang, Wang, & Wei, 2013), QuickFuzz (Grieco, Ceresa, & Buiras, 2016), and LangFuzz (Holler, Herzig, & Zeller, 2012) are focusing on generating fuzzing files for web browsers engines like Mozilla, IE, and Firefox. They use user input format XML, CSS, HTML, PHP, and JavaScript particularly to extract the grammars from those languages. Then, they used them to mutate the generated user inputs. Moreover QuickFuzz (Grieco, Ceresa, & Buiras, 2016), in addition to web browser engines, it focuses on generating fuzzing files for image processing utilities and file achievers. Sargsyan et al. (2018) stated that SD-Gen focuses on generating fuzzing user inputs on programming languages such as C, C++, Python, Java, etc.

According to Guo et al. GramFuzz (2013) has found 36 vulnerabilities that are believed to be severe security issues in IE, Mozilla, and FireFox. MGF (Koroglu & Wotawa, 2019) has found to have an increased code coverage by 8.5% than the mutated fuzzer without using grammars. Also, the MGF found important bugs in the compiler under test (Koroglu & Wotawa, 2019). The experiment results of the fuzzing tools SD-Gen (Sargsyan, et al., 2018), BlendFuzz (Yang, Zhang, & Liu, 2012) and QuickFuzz (Grieco, Ceresa, & Buiras, 2016) showed increased code coverage and more vulnerabilities has been discovered in the tested programs. In LangFuzz, it has discovered 164 bugs in popular JavaScript engines and detected 20 bugs on PHP engine (Holler, Herzig, & Zeller, 2012).

3.2 Grammar-based Fuzzers Guided by Machine Learning

Some other grammar-guided fuzzing tools employ machine learning techniques to produce well-organized and well-formed user inputs which will be able to increase code coverage and discover new bugs.

Learn&Fuzz (Godefroid, Peleg, & Singh, 2017), Skyfire (Wang, Chen, Wei, & Liu, 2017), and GANFuzz (Hu, Shi, Huang, Xiong, & Bu, 2018) tools employ machine learning techniques to generate user input grammars. Jitsunari et al. (2019) proposed a generative model to learn the input token to generate new fuzzing inputs. Godefroid et al. (2017) stated that Learn&Fuzz learns the input grammar and generates fuzzing inputs for fuzzing purposes. Jitsunari et al. (Jitsunari & Arahori, 2019) stated that their study overcomes the problems in Learn&Fuzz (Godefroid, Peleg, & Singh, 2017). Wang et al. (2017) stated that Skyfire use sample inputs and the provided

grammars to generate fuzzing inputs by learning PCSG (Probabilistic Context-Sensitive Grammar) which is based on semantics and syntax. GANFuzz (Hu, Shi, Huang, Xiong, & Bu, 2018) utilizes machine learning (deep learning) to learn the input grammars for network protocol.

Learn&Fuzz (Godefroid, Peleg, & Singh, 2017) focused on PDF files to find grammar and use it for fuzzing generations to test programs. Jitsunari et al. (2019) focused on using a generative model that learns the sequence of tokens in PDF files and generate new fuzzing files. Moreover, Skyfire (Wang, Chen, Wei, & Liu, 2017) focused on input files such as XML, XSLT, and JavaScript to fuzz certain locations in the inputs based on its type that meets the grammar context.

3.3 Grammar-based Fuzzers Guided by Evolutionary Computation

There are some grammar-guided fuzzing tools that employ evolutionary computation methods such as genetic algorithm and genetic programming. These tools take advantage of the crossover and mutation techniques to produce well-organized and well-formed test inputs based on enhanced fitness function.

EvoGFuzz (Eberlein, Noller, Vogel, & Grunske, Sept. 2020), Grammarinator (Hodován, Kiss, & Gyimóthy, 2018), and IFuzzer (Veggalam, Rawat, Haller, & Bos, 2016) utilize grammars in generating fuzzing input files by using evolutionary computation techniques such as mutation and recombination. EvoGFuzz (Eberlein, Noller, Vogel, & Grunske, Sept. 2020) focused on fuzzing JSON parsers, CSS and JavaScript parsers. Grammarinator (Hodován, Kiss, & Gyimóthy, 2018) and IFuzzer (Veggalam, Rawat, Haller, & Bos, 2016) focused on fuzzing JavaScript engines and Grammarinator found over 100 issues in some of JavaScript projects (Hodován, Kiss, & Gyimóthy, 2018). IFuzzer discovered 40 bugs in Mozilla (Veggalam, Rawat, Haller, & Bos, 2016). EvoGFuzz (Eberlein, Noller, Vogel, & Grunske, Sept. 2020) found 11 exceptions more than baseline that found only 6 exceptions. Moreover, EvoGFuzz (Eberlein, Noller, Vogel, & Grunske, Sept. 2020) is significantly higher in code coverage than the baseline.

3.4 Grammar-based Fuzzers Guided by Coverage Feedback

Coverage-guided fuzzing tools are able to effectively create inputs for programs with structured input languages. Coverage-guided grammar-based fuzzing tools can mutate inputs in a small group by using the given grammar (Blazytko, et al., 2019). Moreover, the fuzzing tool has the ability to reasonably merge inputs that lead to critical attributes with a high probability of discovering more risky actions. However, the grammars need human labor, expert knowledge, and hard work to be manually written to obtain correct input format. Also, it is error prone, which makes it difficult to manually provide a correct specification (Blazytko, et al., 2019).

There are several tools that employ this technique; they are NAUTILUS (Aschermann, et al., 2019), AFLSMART (Pham, Böhme, Santosa, Caciulescu, & Roychoudhury, 2019), Pythia (Atlidakis, Geambasu, Godefroid, Polishchuk, & Ray, 2020), and Superion (Wang, Chen, Wei, & Liu, 2018). They combine grammar and coverage feedback to guide fuzzing to explore bugs and vulnerabilities deeply in the program under test. NAUTILUS (Aschermann, et al., 2019), AFLSMART (Pham, Böhme, Santosa, Caciulescu, & Roychoudhury, 2019), and Superion (Wang, Chen, Wei, & Liu, 2018) use the provided grammar to generate new fuzzing input files. Pythia (Atlidakis, Geambasu, Godefroid, Polishchuk, & Ray, Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations, 2020) extract grammars from sample input files to use the grammars for generating fuzzing input files. They use grammars with coverage feedback to mutate interesting inputs and increase the chance to find bugs located deep in the program under test.

NAUTILUS (Aschermann, et al., 2019) and Superion (Wang, Chen, Wei, & Liu, 2018) focus on fuzzing web browser engines such as IE, PHP engine, and JavaScript programs by using XML and JavaScript grammars. AFLSMART (Pham, Böhme, Santosa, Caciulescu, & Roychoudhury, 2019) focuses on AVI and WAV files. It uses their grammar to produce new fuzzing files to test

programs. Pythia focused on extracting grammars from web services like HTTP (Atlidakis, Geambasu, Godefroid, Polishchuk, & Ray, 2020).

NAUTILUS (Aschermann, et al., 2019), AFLSMART (Pham, Böhme, Santosa, Caciulescu, & Roychoudhury, 2019), and Superior (Wang, Chen, Wei, & Liu, 2018) discover bugs in the programs that they tested. NAUTILUS (Aschermann, et al., 2019) has found 13 new bugs in 4 targets mRuby, Lua, PHP, and ChakraCore. AFLSMART (Pham, Böhme, Santosa, Caciulescu, & Roychoudhury, 2019) has found 42 bugs in multimedia open-source programs and 22 of them assigned CVEs. Pythia (Atlidakis, Geambasu, Godefroid, Polishchuk, & Ray, 2020) tested web services APIs GitLab and Mastodon programs for 24 hours and found 29 bugs in them. Superior (Wang, Chen, Wei, & Liu, 2018) 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 (Ruderman, 2007), AFL found only 6 bugs of 34 and JsFunFuzz did not discovered anything (Wang, Chen, Wei, & Liu, Superior: Grammar-Aware Greybox Fuzzing, 2018).

4. DARPA CGC DATASET

DARPA Cyber Grand Challenge (CGC) is a competition developed to spur the research on automated vulnerability discovery and patching. CGC representatives created a dataset containing vulnerable programs with details of explaining a program process. DARPA designed the DARPA Experimental Cyber Research 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 (Shoshitaishvili, et al., 2016). 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 (Shoshitaishvili, et al., 2016).

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.

5. 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 analysis, file reading and searching programs in the dataset. Each program has its own way to interact with the program, 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. These network traffic data can be treated as valid sample input files, and a grammar can be learned from these files and be used in the later process.

This section discusses how to analyze input files from the DARPA CGC dataset to extract grammars. After extracting grammar from the sample input, the grammar 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 1 it 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 1, commands are the first word of each line such as *copy*, *list*, *show*, *erase*, *make*, *last*, *write*, etc.. Parameters are the items that come after a command. Using the first line in Listing 1 as an example, *copy* is a command and the parameters are *README.txt*, and *lcggldeuauqsrx*. On the second line, *list* is a command and no parameter after it. The grammar analysis step is to go through each sample input files for a program and analyze 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 in sample input files, 2) analyze real commands, 3) analyze common parameters, and 4) analyze numbers.

5.1 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 flexibles, that the tool can find repeated or common 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 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.

```

copy README.txt lcggldeuauqsrx
list
show README.txt
erase authentication.db
make gtims
last README.txt 10
show lcggldeuauqsrx
show README.txt
list
write authentication.db
show README.txt
last README.txt 10
last README.txt 10
first README.txt 10
last lcggldeuauqsrx 10
first authentication.db 10
list
logout
evjfrtm 58932
xboskhi 18012
ahpoyva 1614
last authentication.db 10
make flzzofveqexsfje
make kzoycqepdp
write flzzofveqexsfje
chwvsfnspychdjrbyfjunhvlbypwibjeiuxgoipvxr
write authentication.db
erase README.txt
first lcggldeuauqsrx 10
perms authentication.db 2
list

```

```

make zmhvbqsjmzo
perms README.txt 2
last authentication.db 10
first flxzzofveqexsfje 10
perms authentication.db 1
erase flxzzofveqexsfje
last zmhvbqsjmzo 10
perms zmhvbqsjmzo 1
perms README.txt 3
erase gtims
perms authentication.db 4
write authentication.db
exit

```

LISTING 1: Filesystem_Command_Shell Program User Input Example.

5.2 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.

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 commands only occurred for 2 or 3 times. Then the low-occurrence commands are less likely to be the real command, and they should be ignored. Otherwise, the number of real commands will be too high and it takes much longer time to analyze them. In addition, many of the low-occurrence commands are random strings and they are not real commands.

The percentage number is for comparisons to decide whether a word is a real command or not. If the occurrence number of a command is larger or equal to the percentage number, then it will be treated as a real command; otherwise, it will be dropped. The tool currently uses 10% of the number of input files as the percentage number, but the number can be changed easily if needed. For example, if input files count 1000, then the percentage is 100. After counting an item, if the counting number is larger than or equal to 100, the tool considers it as a real command. 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.

```

['ahpoyva1', 'chwvsfnnspsychdjfrbyfjunhvlbypwibjeiuixgoipvxr0
', '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',
'xboskh11']

```

FIGURE 1: Sorting commands.

Using the input in Listing 1 as an example, after the commands are collected, the command list are taken and sorted as shown in Figure 1. Then, for each real command in the list, the number

of occurrences is counted (see Figure 2). After that, it is compared with the percentage number; therefore, the tool keeps the ones that equal or exceed the percentage number. For example, the real commands list for Listing 1 is shown in Listing 2.

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

FIGURE 2: Real commands.

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

LISTING 2: Real commands

In the command list, an extra digit is added after each command. Because the tool needs to distinguish for each command how many parameters it has, a digit is added after each command. By having this digit, the tool knows how many parameters that come after a command and collect them for finding common parameters (next step). As shown in Listing 1, using first line *copy README.txt lcgfldeuausqsrx* as an example, the number of the split words on this line is 3, which means there are 2 parameters following the command. Then it will show a *copy2* on the real command list if it exceeds the percentage number so the tool knows that the command *copy* has 2 parameters. However, if the command does not have a parameter, a digit 0 is appended to the command to indicate no parameter follows the command, such as *list0* in the example. To lower the amount of generated fuzzing files, the max parameter number is set for 3. This means if a line has more than 4 split words, it will be excluded from the analyzing.

5.2 Analyzing Real Parameter

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. The tool manages all the parameters and identifies the parameters that appear mostly with a command if the counted number equals to or exceeds a calculated percentage.

Similar to the percentage number used for determining real command, another percentage number (10%) is used. This percentage 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 the possible parameter occurs more than or equal to the calculated percentage, it is considered as a real parameter for that command. For example, if a real command number occurrence is 200, the tool multiplies it by 10%, which will give a calculated percentage of 20. Therefore, if the possible first parameter occurs 20 times or more, it will be considered as a first real parameter for that command.

```
{'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']]}
```

LISTING 3: Real commands and their real parameters.

If a real parameter was 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.

After that, the new findings for real commands with their real parameters are kept for further processing. The final findings from Listing 1 are shown in Listing 3. On lines 2 and 4, *'erase1'* and *'first2'* are commands. The digit in the end of each command tells how many parameters that follow the command. 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.

5.3 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 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

```
{'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']],
'show1': [['README.txt'], ['authentication.db']],
'write1': [['README.txt'], ['authentication.db']]}
```

LISTING 4: Real commands and their real parameters after changing numbers to [0-9].

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 changed to r[0-9], r[0-9], r[0-9], 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 4 will be obtained. The only changes from previous listing (Listing 3) are on line 10 in Listing 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. Another reason to change numbers in parameters is that the fuzzing tool will substitute the “[0-9]” to generate random numbers in a hope to discover 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.

6. GENERATING FUZZING FILES

After finishing the sample input files analysis, grammars are extracted, which will support generating fuzzing files. This section discusses how the new fuzzing tool creates and generates

fuzz data 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 need 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 be used 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.

6.1 Lines Replacements

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

6.1.1 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 because the aim is to create many input fuzzing files from one input file so the tool will have a higher chance in triggering a vulnerability in a program. Second, it takes a line from the input file and creates a target line to recognize the line that needs to be replaced with a random string. Third, it starts replacing lines from first line and then goes to the next line by increasing the target line 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. 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. Fourth, the tool works by taking the first line and taking the first word of it which is considered a command. Then, the length of the command is calculated by knowing the number of its parameters because the tool will use this number and concatenate it with the command to check with the grammars and to make the replacement based on the number of parameters. 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 the parameters checking. However, if the first word does not match any command in the command list, then the tool will generate a random string for that line. So, this will support testing programs that do not have real commands and may discover a vulnerability from them.

6.1.2 Creating Fuzzing Line

In a line, the tool will have different number of parameters that come with a command. There are commands with no parameters and one, two, or three parameters. For each one, the line is taken based on if it is a command with 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 is going to be substituted by a random number with a different 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 one, two, or three 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 will be added a random string to it with a random length. The idea is to generate a fuzzing line that could lead to revealing a bug in a program under test.

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]". Therefore, those parameters are 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 one in Listing 4. Thus, the tool will be able to change the "[0-9]" with a random number. 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 r12348579204756380801, r2` or `move r3, r385972939754098840242` or even `move r3, r-328495723975892`. Helped with the grammar, the string can be substituted precisely. In addition, it is more likely to trigger a bug in a program compare to substituting the whole line with a random string which may not meet the format requirement and hence will be rejected by the program.

6.2 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 for the selected element from the split line.

6.2.1 Random Test Numbers

In the beginning, random test numbers 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 that is 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.

After completing the generating process from one of the picked random test number, it will be removed from the list. Then, the generating process will continue for every 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 with 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.

6.2.2 Digit 1: Selecting and Replacing Part of a Line

For selecting and replacing a string, random test numbers list for a command are 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 as follows:

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

6.2.3 Digit 2: String Types To Be Generated

In the picked random test number, the second digit is for 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 then inserting them into a program to trigger a buffer overflow or a bug that leads to crash 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 of using the random test numbers for each command in the fuzzer which will help the tool to create different random string in each round.

6.2.3 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. 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 with different lengths in a hope to find a bug or vulnerability in the program. The tool works by choosing between 0 and 1 randomly. If 0 is selected, the tool will use the first list; otherwise, if 1 is selected, it will use the second list. Therefore, the probability to choose between the two list is 50%. 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 be helpful in finding these mistakes before the production stage.

6.3 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, so as stated above, if the selected item is 0 or 1, it taking the 4th element in either of the two lists. If 0 is selected, the length of the random string will be 32 (from the first list). Otherwise, if 1 is selected the length of the random string will be 2200 (from the second list).

Once the random string is generated, it will be plugged in to 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 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 good amount of fuzzing files generated, and they can be fed to the target program for testing.

7. EXPERIMENTS AND EVALUATION

7.1 Experiments

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.

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”.

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 because it violates memory access. “Segmentation fault” message informs that the program encounters a serious error and the program has a bug or vulnerability. As shown in Figures 3 and 4, when a fuzzing input file crashed a program under test, it is displayed on the terminal that the program has been “CORED” or crashed, and then continue testing next fuzzing files.


```

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 3: Printer Program core dumped example.

```

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 4: Payroll Program core dumped example.

For instance, as shown in Figure 5 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" fed to the program. While running other input files, there was no crash or SIGSEGV message.

```

Running on input: testcases1/127.0.0.1.44713-127.44.224.53.2457_6890.pov
> Error!
> QUIT

Running on input: testcases1/127.0.0.1.44713-127.44.224.53.2457_6891.pov
> 308438741 (0x126266d5)
> Error!
>

Running on input: testcases1/127.0.0.1.44784-127.44.224.53.2457_5190.pov
> Error!
> QUIT

Running on input: testcases1/127.0.0.1.44784-127.44.224.53.2457_5191.pov
> 1983834287 (0x763ee8af)
> Error!
>

Running on input: testcases1/127.0.0.1.44788-127.44.224.53.2457_3780.pov
> ./testscript.sh: line 2: 9366 Segmentation fault (core dumped) ./RRPN < "$file" &>> output1.txt

Running on input: testcases1/127.0.0.1.44788-127.44.224.53.2457_3781.pov
> 0 (0x00000000)
> ./testscript.sh: line 2: 9368 Segmentation fault (core dumped) ./RRPN < "$file" &>> output1.txt

Running on input: testcases1/127.0.0.1.44830-127.44.224.53.2457_7100.pov
> 0 (0x00000000)
> QUIT

```

FIGURE 5: RRPN Program Crash.

7.2 Used Tools In Experiments

We compared our work with these four tools: the latest version of AFL (Zalewski, 2014), MOPT (Lyu, et al., 2019), FairFuzz (Lemieux & Sen, 2018), and AFLFAST (Böhme, Pham, & Roychoudhury, 2017).

AFL (American fuzzy lop) is a state-of-the-art greybox fuzzer (Zalewski, 2014). 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. introduced MOPT (Lyu, et al., 2019) 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. stated that FairFuzz (Lemieux & Sen, 2018) 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 Bohme et al., AFLFAST (Böhme, Pham, & Roychoudhury, 2017) is a graybox fuzzing tool that uses Markov chain knowledge (Norris, 1998). 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.

7.3 Testing Results and Observations

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 (Darpa challenge binaries on multiple os systems, n.d.) who ported DARPA dataset from DEGREE system to Linux. There are 247 programs. We excluded 12 challenges because there are issues with compiling and running those programs. 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 (Zalewski, 2014), MOPT (Lyu, et al., 2019), FairFuzz (Lemieux & Sen, 2018), and AFLFAST (Böhme, Pham, & Roychoudhury, 2017) 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. Table 2 shows the number of crashes of each tool and the approximated time to crash most of programs. Therefore, AFL crashed 45 programs, which is 19.14%. AFLFast crashed 44 programs, which is 18.71%. Moreover, MOPT and FairFuzz crashed 54 programs, which is 22.98%. Our tool crashed 79 programs, which is 33.61% of the programs.

Tool Name	Number of crashes	Approximated Time to crash most programs (minutes)
AFL	45	20-30
AFLFast	44	10-15
MOPT	54	15-20
FairFuzz	54	25-30
Our tool	79	5

TABLE 1: Experimental Results.

Figure 6 shows our fuzzing tool discovered vulnerabilities more than other tools. In the help of grammar, our tool found 34 more than AFL, 35 more than AFLFast, and 25 more than MOPT and FairFuzz. 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 find a vulnerability. 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 average of 5 minutes.

Our 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 type of input formats such as GramFuzz (Guo, Zhang, Wang, & Wei, 2013) and Learn&Fuzz (Godefroid, Peleg, & Singh, 2017). 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 (Wang, Chen , Wei, & Liu, 2017) and MGF (Koroglu & Wotawa, 2019).

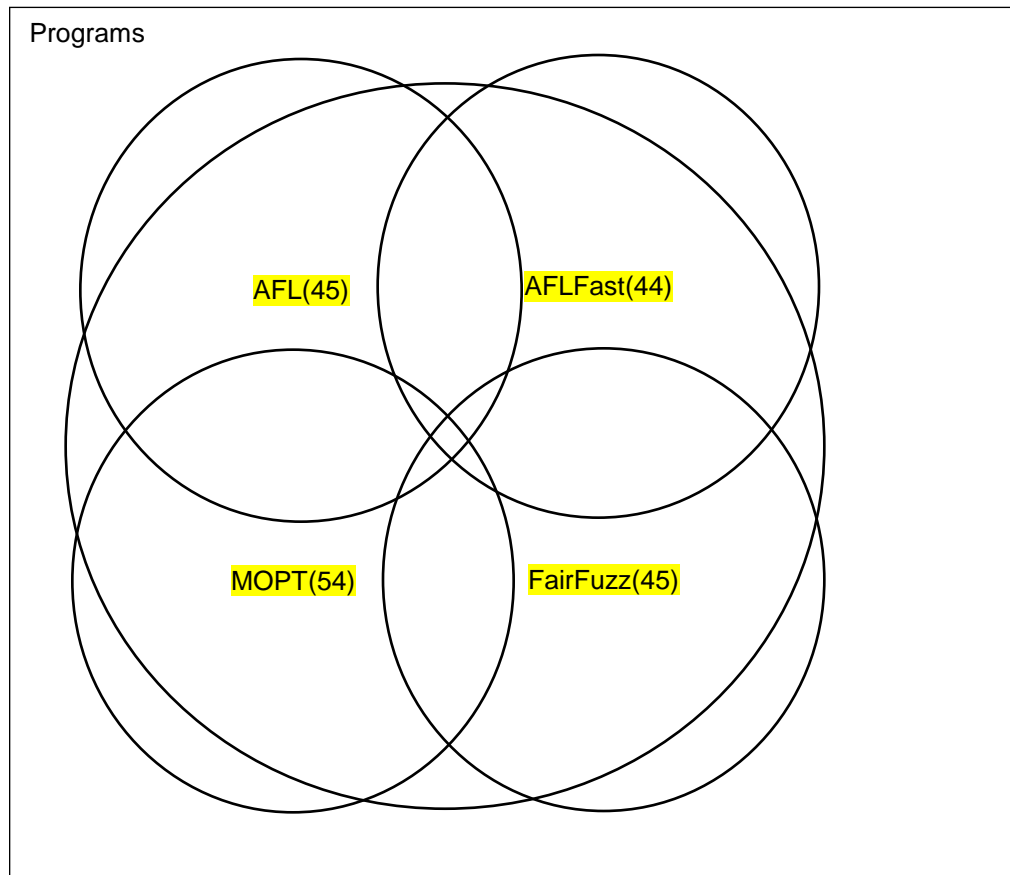


FIGURE 6: Venn Diagram for discovered bugs.

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 the programs. Grammar-based fuzzing support to get the correct format for a program and use it to generate fuzzing inputs that leads to possibly crash a program.

Table 1 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 is showing the number of sample input files for each program. The fourth column is showing 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).

Program name	Generation time (m:s)	# of sample input files	# of fuzzing files	Testing time (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
Filesystem Command Shell	0:30	1000	36627	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

Program name	Generation time (m:s)	# of sample input files	# of fuzzing files	Testing time (m:s)
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_Queueing_Simulator	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
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

Program name	Generation time (m:s)	# of sample input files	# of fuzzing files	Testing time (m:s)
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

TABLE 2: List of Crashed Programs.

8. CONCLUSION AND FUTURE WORK

In conclusion, the research is about to analyze grammars from input files and develop a fuzzer that uses the extracted grammars to generate fuzzing files. Extracting grammars can support the fuzzing tool to generate fuzzing files that may reveal bugs/vulnerabilities in a target program. Also, the fuzzing methods and techniques are more effective to find bugs in the venerable programs. The experiments and testing show that there are 79 programs of the DARPA CGC dataset crashed successfully. The tool extracts critical grammars from sample input files. The grammar is later used when generating fuzzing files to make them more effective. From our experiments, the fuzzer is fast and usually finds vulnerabilities in under 5 minutes. The fuzzing tool can support the software security community with new approach and fuzzing techniques. Moreover, it will give a hand to software developers to find vulnerabilities in their programs before delivery stage. Therefore, the research work could bring in new conceptual methods to the software testing community.

The future work for this research is learning the commands order to support generating new fuzzing files without modifying the sample input files. Moreover, the tool will learn the order of the commands and the probability of the occurrence of each command. Then, it combines them with the analyzed grammars, to generate completely new fuzzing files. This step can be run on top of the current tool. These new fuzzing files will be inserted to the programs to find or discover bugs or vulnerabilities in them. The current plan is to learn commands' order to help further analyze the sample input.

The research learns the structure of the sample input file as well. This allows the fuzzer to generate completely new fuzzing files without mutating or modifying the sample input files. It will give conceptual methods based on inputs' grammar analysis of general programs 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 grammar learning of the sample input files.

9. REFERENCES

Al Salem, H., & Song, J. (June 2019). A Review on Grammar-Based Fuzzing Techniques. *International Journal of Computer Science & Security (IJCSS)*, 13(3), 114-123.

Amini, P. a. (2013, May). Sulley: Pure Python fully automated and unattended fuzzing framework.

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., & Teuchert, D. (2019). NAUTILUS: Fishing for Deep Bugs with Grammars. *NDSS*.

Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., & Ray, B. (2020). Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations.

Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., & Ray, B. (2020). Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *arXiv preprint arXiv:2005.11498*.

Blazytko, T., Bishop, M., Aschermann, C., Cappos, J., Schlögel, M., Korshun, N., . . . others. (2019). GRIMOIRE: Synthesizing structure while fuzzing. *28th USENIX Security Symposium (USENIX Security 19)*, (pp. 1985–2002).

Böhme, M., Pham, V.-T., & Roychoudhury, A. (2017). Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, *45*, 489–506.

Darpa challenge binaries on multiple os systems. (n.d.). Retrieved from <https://github.com/trailofbits/cb-multios>

Eberlein, M., Noller, Y., Vogel, T., & Grunske, L. (Sept. 2020). Evolutionary Grammar-Based Fuzzing. *International Symposium on Search Based Software Engineering*.

Fuzzer, P. (2016). Discover unknown vulnerabilities. *Peach, Peach Fuzzer*. [Online]. Available: <http://www.peachfuzzer.com/>. Accessed on: Jul, 13.

Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., & Chen, Z. (Aug. 2020). GREYONE: Data Flow Sensitive Fuzzing. *Proceedings of the 29th USENIX Security Symposium*.

Godefroid, P. (Jan, 2020). Fuzzing: Hack, art, and science. *Communications of the ACM*, *63*(2), 70-76.

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

Grieco, G., Ceresa, M., & Buiras, P. (2016). Quickfuzz: an automatic random fuzzer for common file formats. *Proceedings of the 9th International Symposium on Haskell*, (pp. 13–20).

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

Helin, A. (2006). Radamsa fuzzer. *Radamsa fuzzer*.

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

Holler, C., Herzig, K., & Zeller, A. (2012). Fuzzing with Code Fragments. *USENIX Security Symposium*, (pp. 445–458).

Hu, Z., Shi, J., Huang, Y., Xiong, J., & Bu, X. (2018). GANFuzz: a GAN-based industrial network protocol fuzzing framework. *Proceedings of the 15th ACM International Conference on Computing Frontiers*, (pp. 138–145).

Jitsunari, Y., & Arahori, Y. (2019). Coverage-guided Learning-assisted Grammar-based Fuzzing. *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

Kim, S. Y., Cha, S., & Bae, D.-H. (2013). Automatic and lightweight grammar generation for fuzz testing. *Computers & Security*, 36, 1–11.

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

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

Liang, H., Pei, X., Jia, X., Shen, W., & Zhang, J. (Sept. 2018). Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3), 1199-1218.

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

Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33, 32–44.

Noller, Y. (Sept. 2018). Differential program analysis with fuzzing and symbolic execution. *ASE 2018: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (pp. 944-947).

Norris, J. (1998, July). Markov Chain (Cambridge Series in Statistical and Probabilistic Mathematics). Cambridge, U.K.: Cambridge Univ. Press.

Oehlert, P. (2005). Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3, 58–62.

Pham, V.-T., Böhme, M., Santosa, A. E., Caciulescu, A. R., & Roychoudhury, A. (2019). Smart greybox fuzzing. *IEEE Transactions on Software Engineering*.

Ruderman, J. (2007). Introducing jsfunfuzz. URL <http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz>, 20, 25–29.

Sargsyan, S., Kurmangaleev, S., Mehrabyan, M., Mishechkin, M., Ghukasyan, T., & Asryan, S. (2018). Grammar-Based Fuzzing. *Ivannikov Memorial Workshop (IVMEM)*, (pp. 32–35).

Serebryany, K. (2015). Simple guided fuzzing for libraries using LLVM's new libFuzzer. *Simple guided fuzzing for libraries using LLVM's new libFuzzer*.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., . . . others. (2016). Sok:(state of) the art of war: Offensive techniques in binary analysis. *IEEE Symposium on Security and Privacy (SP)*, (pp. 138–157).

Veggalam, S., Rawat, S., Haller, I., & Bos, H. (2016). Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. *European Symposium on Research in Computer Security*, (pp. 581–601).

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

Wang, J., Chen, B., Wei, L., & Liu, Y. (2018). Superior: Grammar-Aware Greybox Fuzzing. *arXiv preprint arXiv:1812.01197*.

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

Zalewski, M. (2014). American fuzzy lop. *American fuzzy lop*.

MVC Architecture from Maintenance Quality Attributes Perspective

Safia Nahhas

*Faculty of Computing and Information Technology
King Abdulaziz University
Jeddah 21577, Saudi Arabia*

nahhas.safia@gmail.com

Abstract

This paper provides an explanatory study on MVC (Model-View-Controller) architecture from the perspective of maintenance. It aims to answer a knowledge question about how MVC architecture supports the maintainability quality attributes. This knowledge boosts the potential of utilizing the maintainability of MVC from several sides. To fulfill this purpose, we investigate the main mechanism of MVC with focusing on maintainability quality attributes. Accordingly, we form and discuss MMERFT maintainability set that consists of Modifiability, Modularity, Extensibility, Reusability, Flexibility, and Testability. Besides investigating the mechanism of MVC regarding MMERFT quality attributes, we explain how MVC supports maintainability by examining measures and approaches such as: complexity of code by using a cyclomatic approach, re-engineering process, use of components, time needed to detect bugs, number of code lines, parallel maintenance, automation, massive assignment, and others. Therefore, this paper is dedicated to providing a concrete view of how MVC gets along with maintainability aspects in general and its several attributes particularly. This view helps to maximize the opportunity of taking advantage of MVC's maintainability features that can encourage reconsidering the maintenance decisions and the corresponding estimated cost. The study focuses on maintainability since software that has high maintainability will have the opportunity to evolve, and consequently, it will have a longer life. Our study shows that MVC generally supports maintainability and its attributes, and it is a recommended choice when maintenance is a priority.

Keywords: MVC Architecture, Maintainability, Modifiability, Modularity, Extensibility, Reusability, Flexibility, Testability.

1. INTRODUCTION

The MVC architecture stands for Model, View, and Controller. It separates business logic from presentation functionality (Majeed & Rauf, 2018). MVC provides a kind of an archetype architecture that helps to build applications with several maintainability features (Kazman, Bianco, Ivers, & Klein, 2020). Studies such as (Xu & Liao, 2021), (Gining et al., 2018) and (Akbar & Handriani, 2018) are examples of utilizing MVC's maintainability features. We took benefit from the ISO standard's maintainability attributes that are mentioned in (Bass, Clements, & Kazman, 2013) to constitute the MMERFT maintainability set. The MMERFT maintainability set consists of Modifiability, Modularity, Extensibility, Reusability, Flexibility, and Testability. Accordingly, this paper explores many features of MVC's maintainability through investigating these maintainability quality attributes. In addition to studying most of the maintainability quality attributes that are mentioned in the ISO standard, we investigated the flexibility and extensibility quality attributes with MVC. We found it worthwhile to inspect how MVC supports flexibility, since flexibility is related to adaptive maintenance, as the authors in (Kaur, Kaur, & Kaushal, 2020) pointed out "changes in operation to meet the dynamic needs of any software and adds more flexibility in operation, comes under adaptive maintenance". Also, extensibility is one of the most important advantages of the MVC architecture that facilitates appending new features to the applications and supports adaptive maintenance as well as perfective maintenance (Shekhar, 1998).

The paper provides explanations on how MVC supports maintainability by: 1) investigating the main mechanism of MVC while focusing on MMERFT quality attributes; 2) explaining how MVC supports maintainability by examining measures and approaches such as complexity of code by using a cyclomatic approach, examining how a re-engineering approach can benefit from MVC separated parts, time needed to detect bugs, number of code lines, parallel maintenance, automation, massive assignment, and others; 3) finding and indicating studies and frameworks that touched on maintainability attributes in MVC.

Maintainability is an essential nonfunctional requirement of MVC architecture. Studying MVC architecture with a focus on maintenance matters maximizes the chance of taking advantage of MVC's maintainability. Thus, evolving and lengthening the applications' life. For example, this paper touches on how the MVC architecture can meet the need for code isolation that re-engineering maintenance requires by the separation of concerns that MVC offers. Also, by clarifying that the developer can replace the whole current frontend framework with a totally different frontend framework without any impact on the 'Model part', we can show the high modifiability of MVC. Such examples highlight to which extent MVC architecture facilitates maintainability. Therefore, that can encourage reconsidering the maintenance decisions and the corresponding estimated cost. As the authors indicated in (Kumar, Kumar, & Grover, 2007) and (Molnar & Motogna, 2020), maintenance is one of the major and growing cost concerns.

The rest of the paper is organized as follows: Section 2 presents the methodology of the research. Section 3 examines the MVC architecture and the main maintainability issues. In Section 4, we introduce the MMERFT maintainability set and its quality attributes with MVC. While in Section 5 the discussion of the paper is presented, Section 6 provides the related work. Finally, in Section 7, the paper is concluded.

2. METHODOLOGY

The main aim of this paper is to provide an explanatory study on MVC architecture from the perspective of maintenance, and to answer a knowledge question about how MVC architecture supports the maintainability quality attributes. To achieve this goal, we used the abductive inference and analogic inference designs as illustrated in (Wieringa, 2014). Using abductive inference helped to provide better explanations. We especially focused on architectural explanations. Thus, we analyzed the architectural structure of MVC to understand and analyze the mechanism and capabilities of its components from the maintenance's perspective. To do so, we selected a case study from a population of interest that includes frameworks which adopt the MVC architecture. Accordingly, we selected the Yii framework, which follows the MVC architecture and is used in many real-world applications. The same research could be repeated with other frameworks that follow the MVC architecture. The measures and concepts that we used are based on well-known conceptual framework in software maintenance, such as lines of code, complexity of code, and so on. Data sources included the official Yii website and books such as (Safronov & Winesett, 2014; Winesett, 2012), real Yii applications' source code, project reports and a software engineer. We conducted inductive research since we started by studying and analyzing a case, focusing on its architecture and components, and then came to the conclusion that the MVC architecture supports maintainability with providing explanations.

3. MVC AND THE MAIN MAINTAINABILITY ISSUES

The following issues present the MVC architecture from the perspective of the maintenance.

Re-engineering: Re-engineering facilitates maintaining the software to be adapted to the required changes without affecting its functionality by going through three phases: 1) Reverse engineering, 2) Re-structuring, 3) Forward engineering (Feiler, 1993). Since the reverse engineering as indicated in (Rosenberg & Hyatt, 1996) starts by extracting the requirements and design from the source code, thus the isolation that MVC makes by separating the code parts can facilitate applying this process. Therefore, the reverse engineering process which includes 'slicing or code isolation' can significantly take benefit from the MVC's separated parts.

Lines of codes: one of the issues that affect maintainability is the lines of code (LOC) (Hayes, Patel, & Zhao, 2004). As the authors in (H. Zhang, 2009) pointed out, "Larger modules tend to have more defects". When the number of code lines is less, maintenance will be easier and the chances of error-prone will be less. Figure 1 shows comparing between two pieces of code that do the same thing, which is selecting all rows from the country table. One of the pieces is ordinary PHP code and the other applies MVC by using the Yii PHP framework. We can notice that adopting MVC reduces the number of lines.

Use of components: Using components helps to decrease the cost of maintenance (Algestam, Offesson, & Lundberg, 2002) and gain efficient maintainability (Sharma, Kumar, & Grover, 2007). One of the basic features of MVC is that it supports reusing components. Reusing as indicated in (Seth, Sharma, & Seth, 2009) improves maintainability. The second piece of code (MVC piece) in figure 1 extends the popular model component 'Active Record', which is a basic model component in most MVC applications that reduces the number of lines and improves maintainability.

Complexity of code: The complexity of code is one of the issues that influences maintainability. By using the cyclomatic method, we can compare the complexity of code with following the MVC and without following it. Figure 2 shows the flow graph that is used to compute the complexity of the first piece of code in Figure 1 (code without MVC) as follows: The complexity will be the number of predicate nodes (decision nodes) +1=3. Or it can be computed as the number of edges – number of nodes +2, which will be 8-7+2=3. While the complexity in the second piece (with MVC) is 1. Therefore, we can notice that MVC reduces code complexity, which has a positive impact on maintainability.

```

Ordinary php code:
1 $sql = "SELECT * FROM country ORDER BY name";
2 $result = $conn->query($sql);
3
4 if ($result->num_rows > 0) {
5     // output data of each row
6     while($row = $result->fetch_assoc()) {
7         echo "id: " . $row["id"]." - Name: " . $row["name"]. "<br>";
8     }
9 } else {
10     echo "0 results";
11 }

With applying MVC:
1 $countries = Country::find()->orderBy('name')->all();
  
```

FIGURE 1: Comparing the number of lines between the ordinary PHP code and the MVC code.

Abstraction: A high level of abstraction is one of the MVC's characteristics that boosts maintenance (Morales-Chaparro, Linaje, Preciado, & Sánchez-Figueroa, 2007) and (Gamma, Helm, Johnson, & Vlissides, 1993). As we said, MVC supports reusing components, which is an important step towards providing abstraction. Abstraction releases developers from involving them in rewriting code details and making mistakes. For example, in the Yii framework, due to the MVC separation mechanism, by calling 'findModel(\$id)' built-in function, we will have all the model's data, such as student data, employee data, and so on. Also, 'hasOne(), hasMany()' built-in functions could be used to create the joining relationships rather than go into 'joining issues' details, even though developers could customize these details.

The time needed to detect bugs: as the code gets smaller, as it will be easier to detect errors. As it is shown in figure 1, MVC design was able to reduce the amount of code, which can help to reduce the time needed to detect errors.

Comments providing: Most popular MVC frameworks offer clear comments on their default behavior. This act helps in providing essential comments to clarify the default behavior of various code parts, which can contribute to supporting the maintenance process. In (Chen & Huang, 2009) the authors show that lacking source code comments highly impacts maintainability. Based on their studies, lacking source code comments is the top factor among 10 higher-severity software development problems which affect software maintainability.

Parallel maintenance: MVC, with its separation parts, supports parallelism in team work that includes parallelism in maintenance work. Authors in (Majeed & Rauf, 2018) stated that "MVC patterns provide the facility of parallel development". Similarly, authors in (Rahmati & Tanhaei, 2021) indicated that by using MVC, many changes were made in parallel by multiple teams.

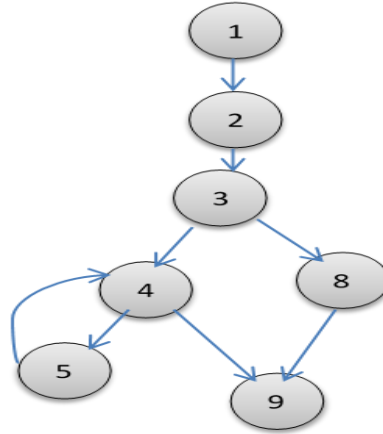


FIGURE 2: The flow graph of ordinary PHP code in the previous figure is used to calculate code complexity.

Massive Assignment: The MVC separation mechanism helps the idea of a 'massive assignment'. For example, massive assignment can populate a model with all user's inputs by just one massive assignment. This approach will make the code cleaner and less error-prone compared with the traditional way. For more clarification, the two pieces of code in figure 3 do the same thing; they both assign the inputs of a form submitted by end-users to the attributes of the 'ContactForm' model (the built-in model in Yii framework). Obviously, the code with massive assignments is easier and shorter.

```

With massive assignment:
$model = new \app\models\ContactForm;
$model->attributes = \Yii::$app->request->post('ContactForm');

Without massive assignment:
$model = new \app\models\ContactForm;
$data = \Yii::$app->request->post('ContactForm', []);
$model->name = isset($data['name']) ? $data['name'] : null;
$model->email = isset($data['email']) ? $data['email'] : null;
$model->subject = isset($data['subject']) ? $data['subject'] : null;
$model->body = isset($data['body']) ? $data['body'] : null;
  
```

FIGURE 3: Comparing the same codes with and without the massive assignment.

Standardization and Identification: The MVC architecture provides clear standardization and identification of its main components (Model-View-Controller), which helps in finding the parts that have problems. This standardization and identification of MVC parts facilitates the ability to do maintenance, especially replacing components with other components without difficulties. For example, replace the current frontend framework with a different frontend framework in the view part without affecting the logic in the model part.

Automation: The main basic codes in MVC frameworks are usually generated automatically. This contributes to reducing errors. For example, in figure 4 we show how the Yii framework provides several generators for models, controllers, modules, and so on.

Iterative development: The separation of concerns that characterizes the MVC architecture is vital in doing iterative development and refactoring, which supports the maintenance process. Authors in (X. Zhang & Zhang, 2016) provide a study on using MVC and an agile iterative approach. More information about the agile iterative approach and modifiability – reusability attributes can be found in the section 'MVC and Modifiability – Reusability'.

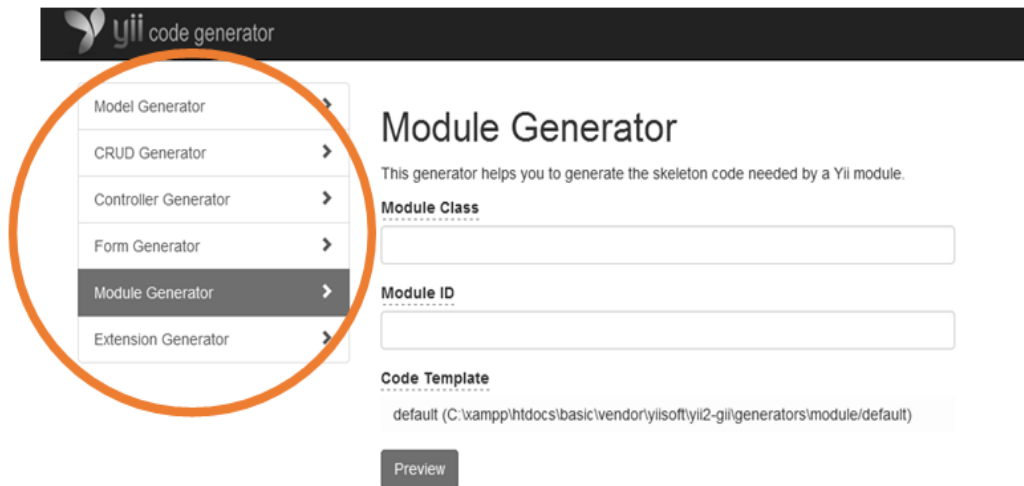


FIGURE 4: Automatic generators in MVC frameworks (Yii framework in this figure).

Integration and verification: Integration test verifies that the components and modules connected successfully. It provides end-to-end test for the behavior of the system. Many popular MVC frameworks provide facilitations to do integration test.

MVC and the maintenance types: mainly, there are three types of maintenance: corrective maintenance to fix and correct system failures; adaptive maintenance to adapt to new requirements; and perfective maintenance to improve performance and efficiency (Alkhatib, 1992). Regarding corrective maintenance, as we will see in the testability section, we can use debugger modules and components since MVC supports modularity and components. For example, in Yii, we have the debugger module and error-handler component to help corrective maintenance. For adaptive maintenance, as we will see in the extensibility section, MVC supports extensibility to extend the software and append to it new changes to make it up to date. For perfective maintenance, MVC frameworks such as Yii take attention to some performance issues such as lazy and eager loading and offer techniques to enhance the performance. For example, rather than using (1 + N) queries in lazy loading, the Yii framework offers eager loading to have just 2 queries in the ordinary relationship.

The next section provides more details on how MVC supports several maintenance quality attributes.

4. MMERFT MAINTAINABILITY SET

In this section, we introduce the MMERFT maintainability set. As we mentioned, we took benefit from the ISO standard's maintainability attributes that are mentioned in (Bass et al., 2013) to constitute the MMERFT maintainability set. As figure 5 shows, the MMERFT maintainability set consists of Modifiability, Modularity, Extensibility, Reusability, Flexibility, and Testability. The next subsections elaborate on how MVC supports each one of these attributes.

4.1 MVC and Modifiability - Reusability

With its separation mechanism, the MVC architecture facilitates the required modifying and reusing in each part without affecting the other. Thus, MVC facilitates modifying the software to meet user requirements. For instance, frontend developers will be able to make their changes without affecting backend developers' work and vice versa. Modifiability relates to reusability, since the developer often needs to modify the old technologies by reusing new ones to meet user requirements. As we indicated, the MVC architecture helps frameworks to be component-based and to reuse components, which facilitates modifiability and reusability. Another point that affirms the reusability of MVC is that it supports DRY (Don't Repeat Yourself) development (WAGHMARE & ADKAR, 2019). Additionally, MVC gets along with the Agile development style. Authors in (Jegarkandy & Ramsin, 2016) mentioned "Overall, based on the evaluation results, the MVC architectural pattern has been deemed as a suitable pattern for use in agile software development". The Agile style of continuous refinements maximizes reuse (Qian, Fu, Tao, & Xu, 2010), and it facilitates the maintenance as the authors in (Prochazka, 2011) confirmed "the usage of agile principles and techniques also brings a lot of benefits to the support and maintenance".

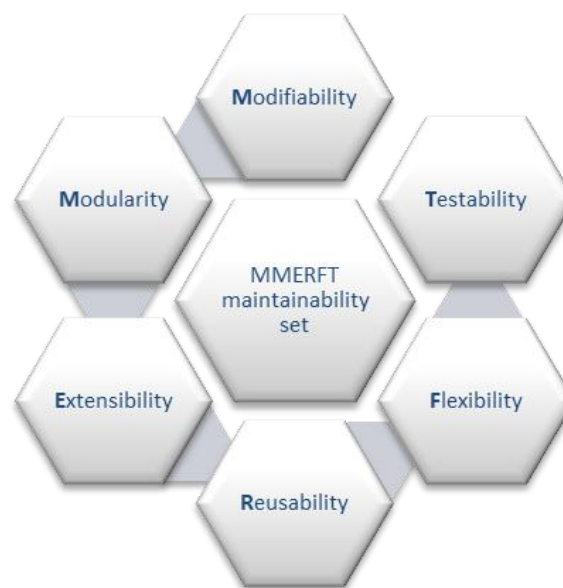


FIGURE 5: MMERFT maintainability set.

MVC in each of its parts simplifies reusing ready-made components, which provide an abstraction that enables developers to reuse and customize the components without the need to look into the details inside them. This feature will minimize the need for writing additional codes, which leads to reducing the probability of mistakes. For example, in the Yii framework regarding the model part, we can use the 'ActiveRecord', which is the main database model component, and reuse the 'ActiveForm' in the view part to build web forms. By reusing the 'ActiveRecord', we can simplify various CRUD manipulations and reduce compatibility issues with various types of databases. The following details give more explanation of the MVC's support for modifiability and reusability in each part.

In Model part: Based on MVC, which assigns a specific part for the data-related logic (Model) and another part for visualizing data (View), that helps make the reuse process based on each part. For example, in the model part, validation rules for the attributes (such as required, string, integer, max and so on) are something that can be reused rather than inventing it each time. Furthermore, these validation rules can be customized and modified based on the requirements. For instance, in the Yii MVC framework, customizing the error message of any rule is available by specifying the message property when declaring the rule as follows:

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Please choose a username.'],
    ];
}
```

In view part: Since MVC separates the business logic of data from the visualization of data, the designer will be able to modify and change the current frontend framework such as the Blueprint CSS framework and reuse other different front-end frameworks such as Bootstrap without any effect on the logic in the model part. Also, modifying and reusing a new theme to rejuvenate the application by using the MVC mechanism becomes very easy. For example, in the Yii framework, it just requires reusing the new layout by putting it in its folder under the view part without any effect on other parts. The same thing goes with reusing and customizing 'Widgets', which is a very important aspect of frontend designing. Widgets are reusable building blocks used in views to create complex and configurable user interface elements in an object-oriented fashion. 'Gridview' is an example of one of the famous widgets in different web frameworks which enables developers to build very interactive tables with pagination, sorting, and so on with just a few lines of code.

In controller part: The controller part in MVC frameworks is usually composed of actions. One of the concepts that shows how MVC architecture facilitates reusability in the controller part is 'Standalone' actions. Standalone actions are usually created to be reused in different controllers or to be redistributed as extensions.

Hence, due to MVC, it has become easier to use famous patterns and components based on each part with a few lines and effective results, such as (ActiveRecord, ActiveForm, Gridview, DataProvider, User Model) which are essential for almost every web application. The straight consequence of reusing will be fewer lines of code, thus fewer errors and bugs. In their research (Cobaleda, Mazo, Becerra, & Duitama, 2016) the authors explained how they used the MVC architecture to provide a reference software architecture for improving the modifiability of personalized web applications. Additionally, in (Otero, 2012) authors confirmed the modifiability and reusability of the MVC architecture and mentioned that MVC is "easy to exchange, enhance and add additional user interfaces".

4.2 MVC and Modularity

Since modules are supposed to have high cohesion and low coupling, modularity is an essential factor in supporting maintainability (Al Dallal, 2009). High cohesion is one of the important metrics in maintainability (Hayes et al., 2004). In many frameworks, MVC is used to build the modules, thus the modules themselves are composites of models, views, controllers, and possibly other components (Safronov & Winesett, 2014). Hence, MVC can help to build the modules based on its architecture, which contributes to supporting modularity and enhancing maintainability. The role of MVC in supporting modularity has been touched upon by authors in (Nitze & Schmietendorf, 2014) and (Grove & Ozkan, 2011).

4.3 MVC and Extensibility

Extensions facilitate adapting the applications to new requirements, which is an important aspect of maintenance (Shekhar, 1998). Extensions can be built as a module or group of modules that can take advantage of the MVC architecture. For example, in the Yii framework, any developer can build an extension based on the MVC module structure, then put it publicly and others will be

able to download it and append it to their applications without any effect on the rest of the application. On the other hand, since every part of MVC is clear, based on the separation that MVC offers, it is easier to append a new feature and put every piece of code in its place, which facilitates the extensibility. For instance, it is extremely easy based on MVC to append the 'Login mechanism' by email to an application by putting every piece of code in its place and taking full advantage of the email login service. Another example is appending to the view part an extension that just converts the frontend direction of the application from left-right to right-left without any affection to the other parts. Many studies agree on the extensibility of MVC, such as (Henttonen, Matinlassi, Niemelä, & Kanstrén, 2007), (Ibrahim, 2009) and (Wang, Guo, & Song, 2009).

4.4 MVC and Flexibility

As we discussed, MVC supports modifiability, modularity, extensibility, and reusability, which leads to more flexibility and high adaptability to new future requirements that the systems may need. MVC is loosely coupled, and it decreases coupling between components by the separation of three distinct areas of concern (Freeman & Sanderson, 2011), which results in more flexibility. To present how MVC supports flexibility, let us take the 'scenarios technique' as an example. Due to the MVC separation mechanism, in the Yii framework a model can be used in different scenarios. For example, the 'User model' could be used in the sign-in process to gather login inputs as a scenario, or it could also be used in signing up for the user registration proposal as another scenario. In different scenarios, a model may use different business rules and logic. A specific attribute might be required in one scenario but not in the other scenario. Therefore, using one model with different scenarios gives the developers more flexibility. Additionally, one model could be used by multiple views and themes, which also shows how MVC helps with flexibility. 'Before and after handling and filters for actions' another embodiment of how MVC supports the flexibility and many options regarding the actions in the control part. Authors in (Cui, Huang, Liang, & Li, 2009) and (Deinum et al., 2012) confirmed the flexibility of the MVC that it is used to build their proposal as well as the early research published in (Gamma et al., 1993).

4.5 MVC and Testability

The separation logic in MVC leads to a facilitated testing process. As well as in (Yaganteeswarudu & VishnuVardhan, 2017), authors indicated that "The MVC framework provides the applications with testability option. Mainly MVC provides unit testable code ". As we explained, MVC supports modularity and components, which make codes more prepared to be tested based on the 'unit testing' approach. Additionally, MVC helps the Test Driven Development (TDD) approach (Zimmerer, 2006), which highly supports testability. The Yii MVC framework, for example, offers unit, function, and acceptance testing and provides ready-to-use test sets for all three test types. Also, it provides modules and components to facilitate testing and detecting errors such as 'Error Handler' and 'Yii Debugger module' which record and display a lot of debugging information, such as log messages, response statuses, the database queries run, and so on. In (Sanderson, 2009) the authors assured the testability of MVC by stating, "Testability MVC architecture gives you a great start in making your application maintainable and testable, because you will naturally separate different application concerns into different, independent software pieces".

5. DISCUSSION

This study has provided illumination and explanation on many aspects and attributes of MVC's maintainability, mainly about: how MVC helps reverse engineering in the reengineering process, and reduces lines of code, complexity of code, and the time needed to detect errors. Also, this study highlighted how MVC helps to provide a high level of abstraction and facilitates use of components, parallel maintenance, standardization and identification, automation, and iterative development. Additionally, we pointed out how MVC serves several types of maintenance, such as corrective maintenance, adaptive maintenance, and perfective maintenance. Besides that, we showed by architectural analytic of the MVC's components mechanism, illustrations of several maintainability attributes, mainly the MMERFT set. The obvious interpretation of all the above outcomes is that MVC has high maintainability and supports the several MMERFT maintainability

quality attributes. Since we studied the Yii framework as a case study that represents other frameworks are built based upon MVC, we can generalize these outcomes and explanations to other similar frameworks with an MVC architecture. Developers and researchers that are interested in a particular maintenance aspect or attribute in the MVC architecture will likely find in this study related explanations, details, and references. While most other studies touched on MVC's maintainability among other features or among other architectures, this study is dedicated to detailing and explaining the MVC architecture cross maintenance aspects and attributes particularly.

6. RELATED WORK

By reviewing current studies related to the paper topic, we found that the existing efforts proposed frameworks or patterns based on MVC and stated the resulting features, including maintainability and other attributes. While this research is designed to study the MVC architecture with a focus on maintainability attributes and aspects specifically, the following studies are examples of research that touched MVC and maintainability among other features or among other architectures. For example, both (Kazman et al., 2020) and (Rahmati & Tanhaei, 2021) provided investigations into maintainability issues regarding several architectures and patterns, including MVC, as a part of their research. In the study (Morales-Chaparro et al., 2007), the authors proposed multiple design patterns based on MVC that could be applied to Rich Internet Applications (RIAs). The authors indicated that MVC improves scalability and maintenance. Eventually, they ended up with an 'RIA deeper MVC' design that increases reusability and decreases maintenance time. In their research (Barrett & Delany, 2004), the authors proposed the 'openMVC framework' based on the MVC architecture. They clarified that their suggested framework is designed in a way that allows several of the basic components in the framework to be modified with no code change or recompile requirement. Accordingly, they confirmed that the redundancy is reduced in their framework. They indicated that their framework improved maintenance" The MVC's Controller object role also consists of two components. The Controller Logic component is independent of the View object role and makes the controller workflow logic more readable and easier to maintain as no layout or style information is interspersed within it". In (Gupta & Govil, 2010), the authors established a framework in the J2EE platform and extended it with XML based on the MVC pattern and multi-tier system. Their multi-tier system includes a presentation layer, a business layer, a data persistence layer, and a database layer to separate codes. They pointed out that with this suggested design, their framework becomes more flexible, expandable, reusable and easy to maintain.

However, none of the prior research that we are aware of are devoted to providing an explanatory study on the mechanism of the MVC design primarily from the viewpoint of maintenance and its quality attributes, as this study does. As we saw above, most current studies built frameworks based on MVC and indicated their features, including maintainability. None of them aimed to study maintainability and its attribute with MVC particularly as this study did. Besides that, this study discussed the MVC architecture with several specific issues closely related to maintenance which have not been examined in the same way in current studies. Furthermore, this paper selected real case to explain the ideas and concepts that can be repeated and generalized with other MVC-based frameworks. Additionally, the study gave concrete examples and perceptions that can be applied to improve maintainability and reduce its cost practically.

7. CONCLUSION

This study focused on MVC architecture from the viewpoint of maintenance. It aimed to answer an explanatory question about how MVC architecture supports the maintainability quality attributes. It worked on revealing how MVC-based systems can take advantage of the MVC's high maintainability. We elaborated how MVC supports maintainability in general and in MMRFT quality attributes particularly. Additionally, we examined other maintainability aspects through examples, measures, and graphics such as: complexity of code by using a cyclomatic approach, the re-engineering process, use of components, abstraction, time needed to detect bugs, number of code lines and others. In addition to reviewing various other studies and MVC-based

frameworks, the Yii framework was selected as a case study to help fulfill the study's objectives. Developers, maintainers, and project managers can have several practical benefits that can lead to extending software's life and evolving its potential. The study helps to highlight various aspects of architecture that could influence maintenance to be taken in consideration during software development. It will help to revise the maintenance decisions and reduce costs. Researchers and developers seeking more knowledge about a certain maintenance feature or attribute of the MVC architecture will likely find related explanations and references in this study. All of that constituted a clear view on how to utilize MVC's maintainability. The overall conclusion that is derived after conducting this research is that MVC generally supports maintainability and its attributes. Therefore, MVC is a recommended choice in applications that make maintenance a priority. Furthermore, this study leads to other future interesting investigations and questions, such as to which extent the MVC architecture will be able to save its maintainability capabilities in large systems. Also, exploring MVC with other quality attributes such as availability, interoperability, reliability and so on is suggested to be conducted. Another future investigation is studying maintainability with other related Model-View- (MV) patterns such as Model-View-View Model (MVVM) and Model-View-Presenter (MVP).

8. REFERENCES

- Akbar, M., & Handriani, I. (2018). *Study and Implementation Information System of Zakat using MVC Architecture*. Paper presented at the IOP Conference Series: Materials Science and Engineering.
- Al Dallal, J. (2009). Software similarity-based functional cohesion metric. *IET software*, 3(1), 46-57.
- Algestam, H., Offesson, M., & Lundberg, L. (2002). *Using components to increase maintainability in a large telecommunication system*. Paper presented at the Ninth Asia-Pacific Software Engineering Conference, 2002.
- Alkhatib, G. (1992). The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice*, 4(2), 83-104.
- Barrett, R., & Delany, S. J. (2004). *OpenMVC: a non-propriety component-based framework for web applications*. Paper presented at the Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters.
- Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice*. In: Pearson.
- Chen, J.-C., & Huang, S.-J. (2009). An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6), 981-992.
- Cobaleda, L.-V., Mazo, R., Becerra, J. L. R., & Duitama, J.-F. (2016). Reference software architecture for improving modifiability of personalised web applications-a controlled experiment. *International Journal of Web Engineering and Technology*, 11(4), 351-370.
- Cui, W., Huang, L., Liang, L., & Li, J. (2009). *The research of PHP development framework based on MVC pattern*. Paper presented at the 2009 Fourth International Conference on Computer Sciences and Convergence Information Technology.
- Deinum, M., Serneels, K., Yates, C., Ladd, S., Vervaet, E., & Vanfleteren, C. (2012). *Pro Spring MVC: With Web Flow*. Apress.
- Feiler, P. H. (1993). *Reengineering: An engineering problem*.
- Freeman, A., & Sanderson, S. (2011). The MVC Pattern. In *Pro ASP. NET MVC 3 Framework* (pp. 63-88): Springer.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). *Design patterns: Abstraction and reuse of object-oriented design*. Paper presented at the European Conference on Object-Oriented Programming.

Gining, R. A. J., Fauzi, S. S. M., Jamaluddin, N. F., Azmi, N. A. M., Hashim, M. A., Fuzi, M. F. M., & Ibrahim, A. F. (2018). Adaptation of. NET MVC Framework in Developing an Agriculture Sources Inventory System. *Journal of Computing Research and Innovation*, 3(3), 6-13.

Grove, R. F., & Ozkan, E. (2011). *The MVC-web Design Pattern*. Paper presented at the WEBIST.

Gupta, P., & Govil, M. C. (2010). MVC Design Pattern for the multi framework distributed applications using XML, spring and struts framework. *International Journal on Computer Science and Engineering*, 2(04), 1047-1051.

Hayes, J. H., Patel, S. C., & Zhao, L. (2004). *A metrics-based software maintenance effort model*. Paper presented at the Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.

Henttonen, K., Matinlassi, M., Niemelä, E., & Kanstrén, T. (2007). Integrability and extensibility evaluation from software architectural models—A case study. *The Open Software Engineering Journal*, 1(1).

Ibrahim, E. (2009). *ASP. NET MVC 1.0 Test Driven Development: Problem-Design-Solution*. Wrox Press Ltd.

Jegarkandy, S. S., & Ramsin, R. (2016). Assessing the suitability of architectural patterns for use in agile software development. In: PATTERNS.

Kaur, A., Kaur, P., & Kaushal, P. (2020). Maintainability Procedure in Component-Based Software. *Journal of Computational and Theoretical Nanoscience*, 17(11), 5156-5161.

Kazman, R., Bianco, P., Ivers, J., & Klein, J. (2020). *Maintainability*.

Kumar, A., Kumar, R., & Grover, P. (2007). An evaluation of maintainability of aspect-oriented systems: a practical approach. *International Journal of Computer Science and Security*, 1(2), 1-9.

Majeed, A., & Rauf, I. (2018). MVC architecture: a detailed insight to the modern web applications development. *Peer Review Journal of Solar & Photoenergy Systems*, 1(1), 1-7.

Molnar, A.-J., & Motogna, S. (2020). A Study of Maintainability in Evolving Open-Source Software. *arXiv preprint arXiv:2009.00959*.

Morales-Chaparro, R., Linaje, M., Preciado, J., & Sánchez-Figueroa, F. (2007). MVC web design patterns and rich internet applications. *Proceedings of the Jornadas de Ingenieria del Software y Bases de Datos*, 39-46.

Nitze, A., & Schmietendorf, A. (2014). *Modularity of javascript libraries and frameworks in modern web applications*. Paper presented at the Selected Topics to the User Conference on Software Quality, Test and Innovation.

Otero, C. (2012). *Software engineering design: theory and practice*: CRC Press.

Prochazka, J. (2011). Agile support and Maintenance of IT services. In *Information Systems Development* (pp. 597-609): Springer.

Qian, K., Fu, X., Tao, L., & Xu, C.-w. (2010). *Software architecture and design illuminated*: Jones & Bartlett Learning.

Rahmati, Z., & Tanhaei, M. (2021). Ensuring software maintainability at software architecture level using architectural patterns. *AUT Journal of Mathematics and Computing*, 2(1), 81-102.

Rosenberg, L. H., & Hyatt, L. E. (1996). Software re-engineering. *Software Assurance Technology Center*, 2-3.

Safronov, M., & Winesett, J. (2014). *Web application development with Yii 2 and PHP*: Packt Publishing Ltd.

Sanderson, S. (2009). *ASP. Net MVC Framework Preview*. Apress.

Seth, K., Sharma, A., & Seth, A. (2009). Component selection efforts estimation—a fuzzy logic based approach. *International Journal of Computer Science and Security, (IJCSS)*, 3(3), 210-215.

Sharma, A., Kumar, R., & Grover, P. (2007). Managing component-based systems with reusable components. *International Journal of Computer Science and Security*, 1(2), 52-57.

Shekhar, S. (1998). Perfective Maintenance & Extensibility. Retrieved from <http://www-users.cs.umn.edu/~shekhar/5180/notes/ch13.html>

WAGHMARE, D. V., & ADKAR, P. (2019). Agile Development using Ruby on Rails Framework.

Wang, Y., Guo, C., & Song, L. (2009). *Architecture of E-Commerce Systems Based on J2EE and MVC Pattern*. Paper presented at the 2009 International Conference on Management of e-Commerce and e-Government.

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer.

Winesett, J. (2012). *Web Application development with Yii and PHP*. Packt Publishing Ltd.

Xu, B., & Liao, Y. (2021). *The Online Exam System Research Based on the MVC Framework*. Paper presented at the 2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC).

Yaganteeswarudu, A., & VishnuVardhan, Y. (2017). *Software application to prevent suicides of farmers with asp. net MVC*. Paper presented at the 2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence.

Zhang, H. (2009). *An investigation of the relationships between lines of code and defects*. Paper presented at the 2009 IEEE International Conference on Software Maintenance.

Zhang, X., & Zhang, Z. (2016). *Research on Web Development Paradigm based on MVC Pattern and Agile Development Mode*. Paper presented at the 2016 5th International Conference on Advanced Materials and Computer Science (ICAMCS 2016).

Zimmerer, P. (2006). *Insights in Real Test-Driven Development*. Paper presented at the Conference of the Association for Software Testing (CAST), Indianapolis, Indiana, USA.

INSTRUCTIONS TO CONTRIBUTORS

The *International Journal of Computer Science and Security (IJCSS)* is a refereed online journal which is a forum for publication of current research in computer science and computer security technologies. It considers any material dealing primarily with the technological aspects of computer science and computer security. The journal is targeted to be read by academics, scholars, advanced students, practitioners, and those seeking an update on current experience and future prospects in relation to all aspects computer science in general but specific to computer security themes. Subjects covered include: access control, computer security, cryptography, communications and data security, databases, electronic commerce, multimedia, bioinformatics, signal processing and image processing etc.

To build its International reputation, we are disseminating the publication information through Google Scholar, J-Gate, Docstoc, Scribd, Slideshare, Bibsonomy and many more. Our International Editors are working on establishing ISI listing and a good impact factor for IJCSS.

The initial efforts helped to shape the editorial policy and to sharpen the focus of the journal. Started with Volume 15, 2021, IJCSS is appearing with more focused issues. Besides normal publications, IJCSS intend to organized special issues on more focused topics. Each special issue will have a designated editor (editors) – either member of the editorial board or another recognized specialist in the respective field.

We are open to contributions, proposals for any topic as well as for editors and reviewers. We understand that it is through the effort of volunteers that CSC Journals continues to grow and flourish.

IJCSS LIST OF TOPICS

The realm of International Journal of Computer Science and Security (IJCSS) extends, but not limited, to the following:

- Authentication and authorization models
- Computer Engineering
- Computer Networks
- Cryptography
- Databases
- Image processing
- Operating systems
- Programming languages
- Signal processing
- Theory
- Communications and data security
- Bioinformatics
- Computer graphics
- Computer security
- Data mining
- Electronic commerce
- Object Orientation
- Parallel and distributed processing
- Robotics
- Software engineering

CALL FOR PAPERS

Volume: 15 - Issue: 6

i. Submission Deadline: October 31, 2021

ii. Author Notification: November 30, 2021

iii. Issue Publication: December 2021

CONTACT INFORMATION

Computer Science Journals Sdn Bhd

B-5-8 Plaza Mont Kiara, Mont Kiara

50480, Kuala Lumpur, MALAYSIA

Phone: 006 03 6204 5627

Fax: 006 03 6204 5628

Email: cscpress@cscjournals.org

CSC PUBLISHERS © 2021
COMPUTER SCIENCE JOURNALS SDN BHD
B-5-8 PLAZA MONT KIARA
MONT KIARA
50480, KUALA LUMPUR
MALAYSIA

PHONE: 006 03 6204 5627

FAX: 006 03 6204 5628

EMAIL: cscpress@cscjournals.org