A Review on Grammar-Based Fuzzing Techniques

Hamad Al Salem *alsa5294@vandals.uidaho.edu*
Computer Science Department
University of Idaho
Moscow, ID, 83844, USA

Jia Song *jsong@uidaho.edu*
Computer Science Department
University of Idaho
Moscow, ID, 83844, USA

Abstract

Fuzzing has become the most interesting software testing technique because it can find different types of bugs and vulnerabilities in many target programs. Grammar-based fuzzing tools have been shown effectiveness in finding bugs and generating good fuzzing files. Fuzzing techniques are usually guided by different methods to improve their effectiveness. However, they have limitation as well. In this paper, we present an overview of grammar-based fuzzing tools and techniques that are used to guide them which include mutation, machine learning, and evolutionary computing. Few studies are conducted on this approach and show the effectiveness and quality in exploring new vulnerabilities in a program. Here we summarize the studied fuzzing tools and explain each one method, input format, strengths and limitations. Some experiments are conducted on two of the fuzzing tools and comparing between them based on the quality of generated fuzzing files.

Keywords: Fuzzing, Grammar-based, Generation, Mutation, Techniques, File Input Quality.

1. INTRODUCTION

Fuzzing is an automatic technique that supports discovering vulnerabilities and weaknesses in a target program by using malformed inputs data from files, network protocol, etc. [11]. Fuzzing has become more important in recent years because of its contributions in software security discipline. Many software companies like Microsoft, Google, etc. are interested in fuzzing and developing fuzzing tools to test the security of software and programs [7].

The fuzzing idea is centered on generating a large number of invalid or bad inputs and feeding them to the target program to trigger errors or cause crashes. Therefore, many studies have been conducted on this topic to find effective ways to generate malformed inputs to trigger vulnerability in a software [8]. Fuzzing could be guided by different techniques, such as symbolic execution, taint analysis and grammar analysis. Each technique brings different advantages and disadvantages to fuzzing. For example, grammar-based techniques usually take shorter time and less resources to generate fuzzing files compared to symbolic execution. However, the drawback of grammar-based techniques is that they heavily rely on sample input files. Moreover, the quality of test inputs is the main point for any fuzzing tools which impacts the efficiency and effectiveness of the tool. There are two main methods for fuzzing tools to generate test inputs mutation and generation [7]. Mutation-based which works by arbitrarily mutating well-formed input files or using other formerly well-defined mutation approaches based on the target program's information that collected during execution. On the other hand, generation method generates new test inputs from a grammar rule or a specification. Moreover, test inputs in fuzzing are usually valid that satisfy the specification to pass the parsing stage and invalid to discover vulnerabilities deeply in the target program.

Fuzzing methods can be three grouped into main types: black box, white box, and gray box [7]. Black box is a way of random testing which does not require any knowledge of the internal code of the target program and it can use some defined rules to mutate any given input data to generate fuzzing input. There are many studies using grammar technique with a black box fuzzer to guide fuzzing combined with other techniques such as [4], [14], and [17]. Based on generation method, inputs must be accepted by some rules or specifications or they will be rejected. White box is a technique that requires the knowledge of internal code of the target program. Moreover, a grammar used with white box fuzzer [13] leverage grammar rules to discover bugs and improve code coverage. Grey box is in the middle between black box and white box which requires the partial knowledge of the target program. It can support code coverage of the program to find bugs faster. Grammar-based fuzzing is effective to generate many input files automatically on the condition that the input format of a program is specific and accurate [7]. This literature review aims at providing a great insight of fuzzing tools that are grammar-based and the used fuzzing techniques to guide fuzzing tools.

2. BACKGROUND

Fuzzing is an easy, flexible, and effective technique to discover software's vulnerabilities before production. By concentrating on grammar-based fuzzing, fuzzing tools use some techniques such as mutation, machine learning, and evolutionary computing for fuzzing guidance. In addition, some machine learning fuzzers like [13] is supported by grammar to generate input tests automatically to increase bug finding. Moreover, evolutionary computing fuzzers like [16] are using grammar to get diverse input tests which lead to code coverage and reveal deep bugs.

The concept of the mutation technique is collecting data (files, network packets, texts, etc.) then modifying or manipulating them randomly or based on some strategies [3]. An Example of the strategies is replacing small text with larger text or changing the length value with larger or smaller value and changing a number with a larger or a smaller number. The advantage of mutation method is it requires little or no knowledge about a target program. However, it may fail the input which depends on challenge response such as checksum.

Machine learning is another method that grammar-based fuzzing is using in some available tools. For example, neural network [13] can support learning grammar of inputs from large corpus. For example, a fuzzing tool must have input files (data set) to learn their grammar format first. Second, the fuzzing tool trains generative neural network to learn the grammar format from input files. Then, it generates fuzzing files based on the learned grammar. After that, the result of good learning would generate well-formed inputs which is accepted by a parser. Otherwise, bad learning would generate ill-formed inputs that will be rejected. Therefore, neural network can help in generating good inputs are able to increase the code coverage in a target program [13]. In learning probability grammar, it takes sample of inputs and its grammar and learns the rules and format model for the inputs and take a probability for a production rule to generate new inputs with variations of grammar structure. The advantage of machine learning approach is generating large diverse test input but better learning does not mean better fuzzing [13] which means the generated better learned input files do not guarantee better fuzzing results.

Evolutionary computing is another technique that grammar-based fuzzing is using to get optimized test inputs to find deep vulnerabilities. Evolutionary algorithms use Darwin theory in biology evolution [5] which provide population of individuals' pressure by environment that causes the natural selection (keeping the fittest) which leads to increased fitness of population. According to Eiben and Smith in [1] Evolutionary algorithm is processed by representing individuals and using the fitness function as evaluation of the individuals. The higher fitness is the best. After that, the individuals that have higher fitness are selected for recombination and mutation to create new individuals. Then, the new fittest individuals are added to the next generation. The iteration continues to enhance the fitness of all individuals. In grammar-based fuzzing, input files are generated using the grammar, and they are considered individuals and represent them by parse tree. Then, the algorithm applies parent selection based on the fitness function, recombination, generation of children, and add them to the next generation of population based on the higher fitness. After that, repeating the operation until finding best test input files by the highest fitness value or the limit is reached. Evolutionary computation is effective in generating

variations of well-formed input files that have the ability to discover bugs and security issues in a target program [16].

3. LITERARURE REVIEW

Many scientific researches have been conducted on fuzzing techniques that use grammar to guide fuzzer. Moreover, the studies include these techniques with fuzzing based on grammar. The techniques are mutation, machine learning (e.g. neural networks), and evolutionary computing (e.g. genetic algorithm and genetic programming) to guide fuzzing and improve the ability in finding bugs in a target program.

3.1 Fuzzing Based on Mutation

Some studies used mutation technique in a grammar-based fuzzing. According to Guo et al. [17] GramFuzz which is using the two techniques grammar analysis of inputs and mutation of the inputs structure to fuzz web browsers. GramFuzz obtains initial input file from internet to build the grammar for web files. They are analyzed to build the grammar to get the trees by using Gold Parser, an open source analysis tool. After getting the grammar trees, the nodes are mutated. Then input test cases can be generated. Authors reported that by combining generation and mutation, test cases will be more effective of fuzzing web browsers. GramFuzz has found 36 vulnerabilities which considered severe security in IE, Mozela, and Firfox [17]. However, GramFuzz only works with web files such as HTML, CSS, and JavaScript.

Sargsyan et al. [15] presented SD-Gen which is an automatic structure data generation based on ANTLR grammar that supports grammar rules for more than 120 languages and file formats. It takes the target grammar and input language as inputs. Then, test data (programs) are generated and mutated. SD-Gen can generate programs for compilers, interpreters, and translator testing because it supports generating programs in C, C++, Java, Pyhton, etc. Results showed SD-Gen is able to increase code coverage [15]. However, it can't provide programs semantic correctness.

BlendFuzz [4] is another grammar-based fuzzing tool which is a model-based framework that is effective fuzzing with grammatical inputs. BlendFuzz generates inputs by first building a parser for the target language. Second, it applies the parser to seed set (input) and obtains a parser tree to extract grammar language from the tree structure and check the ordering of grammar elements. Third, mapping between inferring grammar elements and collecting language constructs to make it easy for mutation. Lastly, it uses mutation technique to the string by selecting fragments and replacing another with the same type. These inputs test the target program so the complex structures and program's edges are covered. In the end, after experiment the results showed the approach is effective and enhanced the code coverage and revealed security vulnerabilities [4]. However, it can't generate new grammar components other than those available in the seed set and can't always generate syntactically correct inputs.

QuickFuzz [6] has been introduced by Grieco et al. which leverages Haskell's QuicKChick (the well-known property-based random testing library) and Hackage (the community Haskell software repository) with combining a general purpose bit-level mutational fuzzers (e.g. Radamsa and HonggFuzz) to create fuzzing automatically for some well-known file format without providing any file format specifications. It generates input files based on a grammar then performs mutational technique on these files to trigger unexpected behavior in the target program. QuickFuzz was tried with real world programs and found to be successfully effective in discovering most important vulnerabilities [6]. However, in the start of generating, some randomly derived inputs are not effective in generation source code because it rejected in the parsing stage. Also, there are some issues of using third party's package because some modules do not support certain complex file types.

LangFuzz [2] was introduced by Holler et al. which is a fuzzing tool using blackbox fuzz testing of engines based on grammar. LangFuzz takes the grammar and sample code to learn language fragments and test suite for code mutation. In code mutation, it is divided into two phases: a learning phase and main mutation phase. In learning phase, a group of sample codes are operated with a parser using grammar. The parser will separate the input code sample into code fragments which are non-terminal in the grammar. Once the learning phase is finished, mutation phase starts by selecting some code fragments

and replacing them with others of the same type. Using code generation, step wise expansion is used by considering code as syntax tree. LangFuzz uses code mutation, which is the primary technique, and random generation to generate test cases to test the engine before passing the test cases to the interpreter. By combining two types of code generation (mutation and generation) LangFuzz has found 164 real-world bugs in popular JavaScript engines and 31 security related vulnerabilities and detected 20 bugs on PHP engine [2]. LangFuzz has some issues in the less test cases or biased tests decrease LangFuzz performance. In addition, to use the tool for a new language it needs necessary changes to be compatible and accepted by the new language.

3.2 Fuzzing Guided by Machine Learning Technique

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

Godefroid et al. [13] designed Learn&Fuzz which uses machine learning (Neural Network) to generate input grammars for grammar-based fuzzer automatically. Learn&Fuzz uses neural network-based learning methods to learn a grammar for non-binary PDF data object such as formatted text. It uses input sampling techniques to generate PDF objects from the learned distribution. There are 1300 pdf pages and they are defined by rules. The grammar rules are huge, heavy, and ponderous but they are structured well and adequate for learning with neural network. Learn&Fuzz utilizes learned input probability distribution to guide the tool where to fuzz inputs in a smart way. It shows that the neural network technique is able to generate well-formed inputs and increase coverage of input parser more than different variations of random fuzzing [13]. Nevertheless, Learn&Fuzz is not able to process less structured inputs such as images, videos, and audio files because it is considered for text formatted inputs.

Wang et al. stated that Skyfire [8] is a data-driven seed generation approach. It takes a corpus and grammar as inputs and generates inputs in two steps. First, it parses the collected samples based on the grammar and generates the AST (Abstract Syntax Tree) trees. Then, it learns the PCSG (Probabilistic Context-Sensitive Grammar) that is based on semantics rules and syntax features. Second, it generates seed inputs by looping to select and apply grammar rules that satisfy the context on non-terminal symbols until there is no more non-terminal symbol in the output string. Then, it applies low probability on high probability production rules to obtain uncommon input with variety of grammar structure. Skyfire makes selection on the resulting input seed to filter out the similar seeds to reduce duplication. In the end, Skyfire mutates the selected input seeds by randomly selecting leaf-level node in the trees with the same node with applying semantic rules and remaining grammar structure. The results show that skyfire effectively improves the code coverage and enhances the ability to find bugs [8]. However, it is only limited for files that their format are XML, XSL, and JavaScript.

According to Hu et al. [18] GANFuzz uses machine learning for industrial network protocol to fuzz network protocol in which input test is generated using protocol grammar by creating specifications or reverse engineering from network packets. In this study, by using machine learning (neural network) an automated test input is generated by using deep-learning techniques to learn protocol grammar. After that, GANFuzz takes advantages to train test inputs over the network packets to get protocol grammars then generates invalid input messages that lead to discover some bugs and errors. The results showed that GANFuzz is effective in code coverage and deeply testing [18]. However, there are some limitations such as GANFuzz cannot handle file operations, generate correct syntactically protocol specifications, and graphical user interface errors.

3.3 Fuzzing Guided by Evolutionary Computation Technique

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

Hodován et al. [14] created the fuzzing tool Grammarinator which works with generation and mutation-based fuzzers with help of grammar. Therefore, it uses parser grammar to generate input test cases and build abstract syntax tree for each test case and analyzes them. Moreover, evolutionary algorithm is used

Fuzzing Tools	Input Format	Sample Inputs	Technique	Usage of Grammar	Strengths	Limitations
Learn&Fuzz [13].	PDF format.	Three types: NoSample, Sample, SampleSpace	Grammar-bases using Neural network (machine learning).	Learn the input grammar then generate new test input as PDF files.	Learning well-formed inputs is useful to guide fuzzing where to fuzz the inputs.	Difficult to learn less structure inputs like images.
GramFuzz [17].	HTML, CSS, JavaScriptc.	Web Files from internet.	Grammar-based.	Mutate the original initial test cases.	Generate more effective test cases that effectively discovered new bugs.	Limited for web files.
Grammarinator [14].	Simple formats but customizable for complex formats.	Example grammar.	Grammar-based.	Mutate the new generated test inputs.	Useful in hardening of the programs by discovering different bugs.	Limited for JavaScript engines.
BlendFuzz [4].	String.	Seed set.	Grammar-based.	Build the grammar from seed set then generate new test inputs then mutate them.	Improves code coverage help to find new vulnerabilities.	Cannot generate new Grammar components other than those available in the seed set, cannot always generate syntactically correct inputs.
LangFuzz [2].	No particular language.	Language Grammar, Sample code, and test suite.	Grammar-based Mutation/generation Fuzzing.	Generate mutated new test inputs after getting language grammar from sample code.	Combining two Approaches make LangFuzz successfully discover bugs.	Less test cases or biased tests decrease LangFuzz performance. To use it for a new language needs necessary changes.
Skyfire [8].	XML, XSL, JavaScript, grammar.	XML, XSL, JavaScript files from internet.	Grammar-based Probabilistic context-sensitive grammar.	Generate new test files based on grammar rules and probability grammar learning from sample input files, then mutate them.	Generate well-formed inputs that are useful to improve code coverage and ability in finding bugs.	Limited for XSL, XML, JavaScript.
IFuzzer [16].	JavaScript.	Grammar, Sample code, test suit.	Grammar-based using Genetic programming.	Generate new test code files from grammar and code sample inputs.	Fast to find bugs. Main strength is its feedback loop and the evolution of inputs as dictated by its new fitness function.	Less quality in code generation. For new language or code needs necessary changes.
GANFuzz [18].	Protocol message format.	Grammar, Sample protocol message.	Grammar-based using Neural Network (machine learning)	Generate new test files based on training protocol grammar over protocol message.	Effective in code coverage and deeply testing	Cannot handle file operations, generate correct syntactically protocol specifications, and graphical user interface errors.
SD-gen [15].	Language data.	Target grammar.	Grammar-based.	Generate new data files and mutate them.	Increases the code coverage.	Can't provide programs semantic correctness.

Table 1: Fuzzing Tools Summaries.

for mutation and recombination to the test input files. Grammarinator defines the depth of generated structure and focuses its generation on the less visited parts. It also defines complex actions and decides the correct test cases that the grammar can describe. Grammarinator is used to test different JavaScript engines and is found to be useful, effective, and has found more than 100 new issues [14]. However, it is only fuzzing JavaScript engines.

Veggalam et al. [16] developed a fuzzing tool which is using evolutionary computation techniques such as genetic programming to guide fuzzing called IFuzzer which takes context-free grammar as input to generate test cases by generating parse trees and extracting code fragments from test suit. IFuzzer uses genetic programming technique which utilizes mutation and crossover and leverages the improving fitness function to enhance the quality of generating input (codes) test cases. The results showed that IFuzzer is fast in revealing bugs compared with the state of art fuzzing tools [16]. Moreover, it found 40 bugs in old version of JavaScript interpreter of Mozila and 17 bugs in latest version of the interpreter [16]. However, IFuzzer is less quality in code generation and for new language or new code IFuzzer needs necessary changes.

3.4 Comparing Different Grammar-based Fuzzing Techniques

Table 1 summarizes fuzzing tools based on 7 categories. First column is fuzzing tools which clarifies the tools' names. The format of the input to each tool is presented on column 2. Third column is sample inputs which clarifies what types of sample inputs that the tool starts with at the first stage of fuzzing. The fourth column explains what technique that the tool uses. Fifth column is usage of grammar which explains how the tool uses the grammar technique. The strengths and weaknesses of each tool are discussed in columns 7 and 8.

4. EXPERIMENTS

To experiment the grammar-based fuzzing techniques, we have chosen two fuzzers Skyfire [8] uses machine learning technique to generate fuzzing files and Grammarinator [14] uses evolutionary technique to generate fuzzing files. In this section, we installed both of the fuzzing tools to check the quality of fuzzing files that are generated by them.

4.1 Skyfire

The Skyfire installing experience was not easy in the beginning because the developer added Skyfire software in Github repository without giving enough instruction on how to install and run skyfire. So, we installed it in Ubuntu 18.10 VM, but we couldn't run it from the terminal because there is no scripts commandline to use for running it. We installed Skyfire on Windows 10, then added the source code to IDE eclipse and by following the instructions we use MySql to create a database to connect it with the Skyfire code for learning grammar. Then, the tool ran perfectly.

```
<?xml encoding="utf-8"  version="1.0" ?>
<test>
<xslo:variable test="self::xhtml:pre"
select="seda:Contains/seda:Appraisal"  test="$xml2rfc-ext-strip-vbare='true'" />
<x:elements-xslt>     <vra-p:versionOf>     <xsl:copy>     &#x2203;     <wot:signed>
<x:param select="@dur2" />     &#x39D;     </wot:signed>     <jabberID>
<errorname>  <vra-p:versionOf>     &#x3D2;     &#x3B5;     &#x3B2;      
</vra-p:versionOf>  <w:endnote test="$generate.component.toc != 0" >
&#x2592;     &pt235;     <xsl:value-of test="empty($jpackageDoc)" />
&le;     &olarr;     </w:endnote>  </errorname>     <xsl:copy>
<body>     &pt456;     &#x251c;     &#x0342;     </body>
<seeie test="$mode = 'fit'" >     &pt13456;     &#xF4;
<people>  <axsl:choose>  <vra-p:versionOf>     &phi;     &#x03C1;
&#x202A;
</vra-p:versionOf>    </axsl:choose>    </people>
</seeie>  </xsl:copy>           </jabberID>       </xsl:copy>
</vra-p:versionOf>       </x:elements-xslt>    </test>
```

FIGURE 1: XML Fuzzing File Generated by Skyfire.

Skyfire takes grammar and sample inputs then learns them to generate fuzzing files by following the grammar rules. Skyfire generates well organized fuzzing files as shown in Figure 1 which satisfy the grammar such as XML or JavaScript syntax format that helps to trigger an error in a target program e.g. web browser engines. In addition, Skyfire controls the number of generating fuzzing files by "numOfSamplesToGenerate" variable in the code and it also controls the length of fuzzing file by "maxDerivationDepth" variable. The best thing about Skyfire is that its mutation work is replacing same type of the context randomly by selecting the right and same type from the pool without mixing or scrambling its contents to keep the input structure valid. However, fuzzing files with larger size will be more complex and will not help find any error or will be discarded by some fuzzers because they become useless [8].

4.2 Grammarinator

The experience of installing Grammarinator was easy because the documentation is available and it is specific and precise. We were able to install it on Ubuntu 18.10 and run it by using the provided command line scripts and it works perfectly.

Grammarinator takes grammar as input then processes it to generate unlexer and unparser Python programs. Using the two generated programs it generates fuzzing files by defining the number of generated fuzzing files needed and the size of fuzzing files.

By taking a look at some generated files, the files seem to be unorganized because they are not following the syntax format. As shown in Figure 2, it is difficult to understand the file's content because there are some mixed and scrambled characters, numbers, and symbols, because the type of mutation and recombination that Grammarinator is using which create these random symbols and characters. The file has a lot of white spaces and unknown data which may not help to trigger any errors when targeting a program.

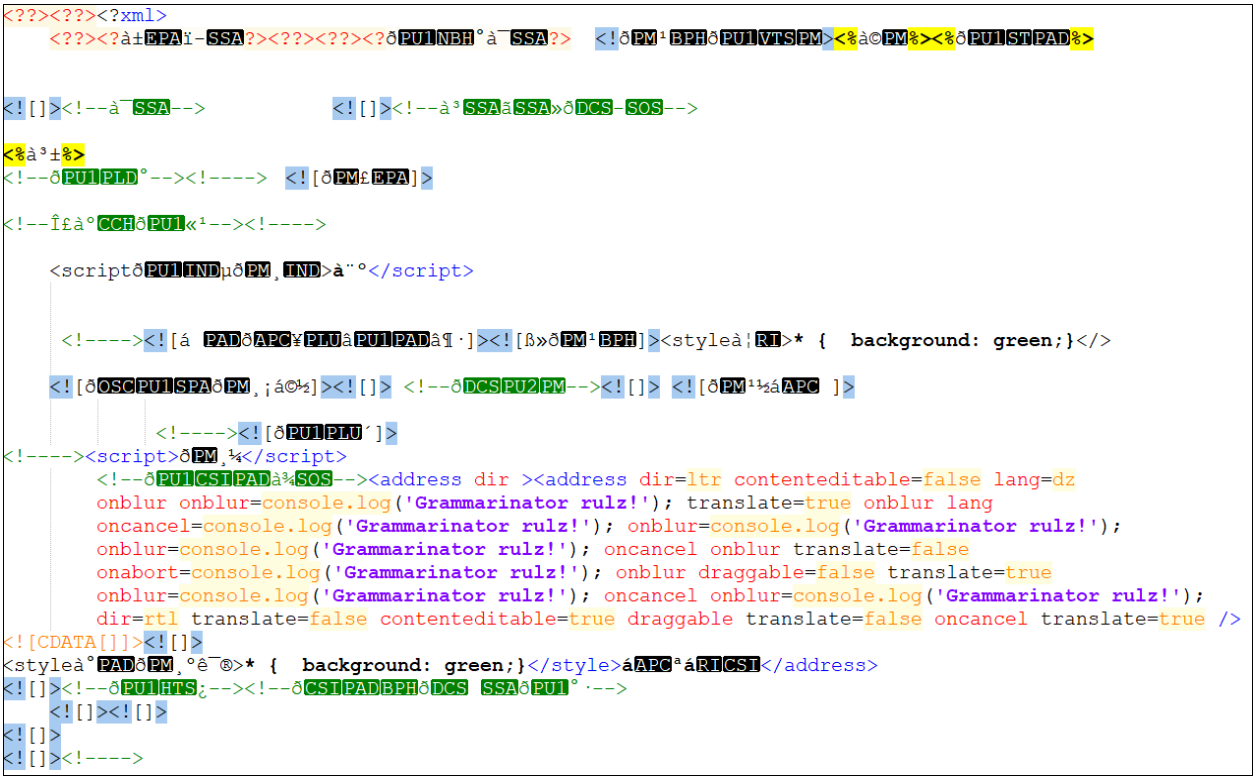

FIGURE 2: XML Fuzzing File Generated by Grammarinator.

5. LESSON LERRNED

Most of the grammar-based fuzzing techniques take certain file format specific input structure. If there is no known input format or specification, the fuzzing tool would face some issues and difficulties to generate and accept test inputs. So, it is important for grammar-based fuzzing tool to have grammar specification which will support in generating valid test input that satisfy the program and test and help to reveal interesting bugs. Moreover, grammar-based fuzzer can increase code coverage and reach deep locations in a program [9].

Most techniques with grammar-based fuzzing is taking grammar and sample input in the first stage of fuzzing to generate test cases after applying fuzzing techniques on the inputs. GramFuzz [17] uses sample input to generate grammar rules then applies them to generate new test inputs. Moreover, mutation is the most common technique used for fuzzing guidance because it is effective in finding deep bugs when using with grammar-based fuzzing tool [17].

The majority of grammar-based fuzzing tools are strict meaning they accept specific kind of input format or specification and fuzz specific type of a target program, engine or application. For example, GramFuzz [17] uses HTML, CSS, and JavaScript as input files and a web browser as a target program. Also, LangFuzz [2] uses programming languages codes such as C++, Java, PHP etc. and an interpreter engine as a target program. Moreover, IFuzzer [16] uses JavaScript code as input and JavaScript engine such as Mozilla's engine as a target program. So, it is difficult on any grammar-based fuzzing tool to have general fuzzing for most of applications and software rather than fuzzing a specific target program because there are many file formats and grammar-based fuzzing uses specific format or specification of input structure.

Grammar-based fuzzing tools generate fuzzing files in a shorter time and use less resources compared to fuzzing tools using symbolic execution or taint analysis. With shorter time a grammar-based fuzzer can generate thousands of fuzzing files. On the other hand, symbolic execution fuzzer takes longer time to generate many fuzzing files and uses more resources because of code interpretation [10].

Some of grammar-based fuzzing tools use a parser tree to mutate randomly or find a way to select interesting paths to discover bugs in unreachable places and increase code coverage in a target program. Others use some methods like package libraries as in QuickFuzz [6] or training inputs to optimize them to be well-formed test input then select some of them randomly and evaluate the code coverage by a parser as in Learn&Fuzz [13]. Moreover, some tools generate inputs based on grammar rules and select interesting ones with no duplication then mutate them as in Skyfire [8].

After finishing the experiments, we came up with some thoughts and ideas. First, Skyfire generates well-formed fuzzing files that follow XML format exactly and they are readable. However, most of the generated fuzzing files can't trigger bugs in the target programs unless the fuzzing files feed to a general purpose fuzzer like AFL [8]. Wang et al. [8] stated that Skyfire improves code coverage, but when we see the code coverage data sets, they built their assumption based on the 5 out of 39 target programs that Skyfire has increased the code coverage and the remaining 34 has not shown any evidence of increased code coverage. Therefore, their conclusion on code coverage improvement is not convinced.

Grammarinator generates fuzzing files with unreadable characters and symbols. The reason of it is that the tool is using mutation and recombination randomly, e.g. bit flipping, bit swapping, byte flipping, byte swapping, etc. which cause the tool to generate these unreadable characters and symbols as shown in Figure 2. Since the purpose of our experiment is the quality of the fuzzing files, we came up with that the generated fuzzing files by Grammarinator are not very effective. Because the random changes affect the default XML format data, which means the target program will most likely reject most of the generated fuzzing files because of not following XML format accurately. Hodovan et al. [14] stated that the tool has discovered over 100 issues without showing data sets as evidence of their discovery even though they did not mention the types of these issues. Moreover, they concluded their case study [14] with the increased code coverage by generating fuzzing files towards less visited paths without providing an accurate percentage number of the code coverage.

6. FUTURE WORK FOR RESEARCH AND DEVELOPMENT

For future direction, we will develop a technique to analyze user inputs to different programs and to detect the grammars of the user input. After that, we will generate fuzzing files based on the detected grammars information, so that the generated fuzzing files can interact with the target program. Many problems can be caused by unexpected user input, because programmers do not always check user input completely, such as input format, symbols, length of the string. By generating and testing different forms of user input, it is possible that vulnerabilities, such as buffer overflow, integer overflow, and format string problems, can be triggered. Grammars will be detected from the sample user input, and they will guide generations of the fuzzing files. By using grammars, more effective fuzzing files can be generated to correctly interact with the program, therefore reach deeper levels of the program.

7. CONCLUSION

Fuzzing has made life easy and flexible for bug and vulnerabilities finding in a software, application, program, interpreter engines, etc. This literature review focuses on grammar-based fuzzing tools which are guided by mutation, machine learning, or evolutionary computing to generate better fuzzing files.

Generating valid inputs which are able to trigger bugs in a program is the main point for high quality fuzzing. Moreover, these inputs must be accepted in the first stage by the grammar to go through fuzzing techniques, or they are rejected. After satisfying the grammar rules, the techniques such as mutation, machine learning (e.g. neural network), or evolutionary computing (e.g. genetic programming) can be applied to help generate fuzzing files.

Grammar-based fuzzing takes shorter time and less resources to generate fuzzing files. However, some limitations include being unable to process some unstructured inputs files like images, not generating correct syntactically input specification, cannot work with undefined grammar specifications for inputs, and limited for specific file format.

8. REFERENCES

[1] Eiben, A. E., & Smith, J. E. (2003). Introduction to Evolutionary Computing. Natural Computing Series. doi:10.1007/978-3-662-05094-1.

[2] Holler, C., Herzig, K., & Zeller, A. (2012). Fuzzing with Code Fragments. Presented as part of the 21st {USENIX} Security Symposium .

[3] Miller, C., & Peterson Z. (2007). Analysis of mutation and generation-based fuzzing. Independent Security Evaluators, Tech. Rep.

[4] Yang, D., Zhang, Y., & Liu, Q. (2012). BlendFuzz: A Model-Based Framework for Fuzz Testing Programs with grammatical inputs. IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, (pp. 1070-1076).

[5] Darwin, C. (2004). On the origin of species, 1859. Routledge.

[6] Grieco, G., Ceresa, M., & Buiras, P. (2016). QuickFuzz: An Automatic Random Fuzzer for Common File Formats. Proceedings of the 9th International Symposium on Haskell.

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

[8] Wang, J., Chen, B., Wei, L., & Liu, Y. (2017). Skyfire: Data-Driven Seed Generation for Fuzzing. IEEE Symposium on Security and Privacy (pp. 579-594). IEEE.

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

[10] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. NDSS.

[11] Oehlert, P. (2005). Violating Assumptions with Fuzzing. IEEE Security & Privacy, 3, 58-62.

[12] Godefroid, P., Kiezun, A., & Levin, M. (2008). Grammar-based Whitebox Fuzzing. ACM Sigplan Notices, (pp. 206-2015).

[13] 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.

[14] 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.

[15] Sargsyan, S., Kurmangaleev, S., Mehrabyan, M., Mishechkin, M., Ghukasyan, T., & Asryan, S. (2018). Grammar-based Fuzzing. Ivannikov Memorial Workshop (IVMEM).

[16] 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).

[17] Guo, T., Zhang, P., Wang, X., & Wei, Q. (2013). GramFuzz: Fuzzing Testing of Web Browsers Based on Grammar Analysis and structural mutation. Second International Conference on Informatics & Applications (ICIA).

[18] 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.