Grammar-based Fuzzing: From Traditional to Probabilistic Methods

Cunhan You
Shanghaitech University
youch2022@shanghaitech.edu.cn

Ziyang Li
Shanghaitech University
lizy5@shanghaitech.edu.cn

Yang Liu
Shanghaitech University
liuyang12022@shanghaitech.edu.cn

Zixian Zhang
Shanghaitech University
zhangzx12022@shanghaitech.edu.cn

***Abstract*—Grammar-based fuzzing is a technique for generating test inputs for software systems based on a formal grammar specification of the input language. Traditional grammar-based fuzzing uses mutation-based, machine learning-based and evolutionary-based methods to generate inputs that are guaranteed to be syntactically valid according to the grammar. However, recent developments have introduced probabilistic methods for grammar-based fuzzing, which allow for the generation of inputs with a certain probability of being syntactically valid. Comparing with the traditional one, probabilistic grammar-based fuzzing has several advantages, including greater flexibility, wider range of test cases, and improved efficiency.**

***Index Terms*—Fuzzing, Grammar-based, Probabilistic, Survey**

I. 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. The concept of fuzzing was first proposed in 1990s [1], and it 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.

The fuzzing idea aims at generating a large number of invalid or bad inputs and feeding them to the target program to trigger errors or cause crashes. Fuzzing could be guided by different techniques, such as symbolic execution, taint analysis and grammar analysis, and each has different advantages. To grammar-based techique, it remarkably shortens executive time by avoiding invalid test data and generating test data automatically, while an iuput file and a grammar with high complexity has to be generated.

To solve the limitation of the quality of the sample input file when modifying sample input files, new tools aiming at learning the commands order, the probability of occurrence of each command, and the structure of sample input file have been put into use in recent years. By learning the order and probability of the commands from sample input files, the new fuzzer can generate fuzzing input files based on that information and the extracted grammar. Not limited by the order of commands in the sample input files, the new fuzzer goes deeper in the program hence may reveal more bugs and vulnerabilities [2].

In this paper, we try to introduce the working principle of the new fuzzer combined with the help of probability method, and how it improves the effectiveness and efficiency of vulnerability discovery. Besides, we show some instances of traditional grammar-based fuzzing techniques and some of the smartest fuzzers. Then, we give an overview of a detailed version of comparison between the new method with probability and those traditional methods in their own advantages and disadvantages. At last, we try to point out some possible future works of how this new technique may develop.

The rest of the paper is organized as follows: "Background" section presents background knowledge on some popular and traditional fuzzing techniques. In "GBF: traditional method" section, we introduce mutation-based, machine-learning based and evolutionary based fuzzing and their related state-of-the-art works. In "GBF: probablistic method" section we discuss how a fuzzer combined with probablistic mathematical methods may fill and improve some corresponding disadvantages against traditional methods. In "Comparison" section, we compare the advantages and disadvantages between probablistic and traditional methods in detail. In "Future work" section, we raise our humble opinions on the possible new trends of fuzzing. And we conclude our paper in "Conclusion" section.

II. Background

In this section, we try to give a perspective on fuzzing, including the basic techniques background knowledge, working process of fuzzing, types of fuzzers and challenges in improving fuzzing.

Fuzzing is now the most popular vulnerability discovery technique [3]. In general, fuzzing starts with generating a large number of normal and abnormal inputs to the target application, and attempts to detect exceptions by feeding the generated inputs back to the target application and monitoring the execution status. Compared with other techniques, fuzzing is easy to deploy and of good extensibility and applicability, and could be performed with or without the source code [4]. The other advantage of fuzzing is that it gains a high accuracy because it is performed in the real execution. Besides, fuzzing requires few knowledge of target applications and could be easily scaled up to large scale applications.

However, fuzzing is also faced with many disadvantages. Two of the most influential disadvantages are low efficiency and low code coverage. Nowadays, the reseachers are trying to overcome these two main disadvantages in the state-of-the-art methods, such as grammar-based fuzzing.

A. Fuzzing

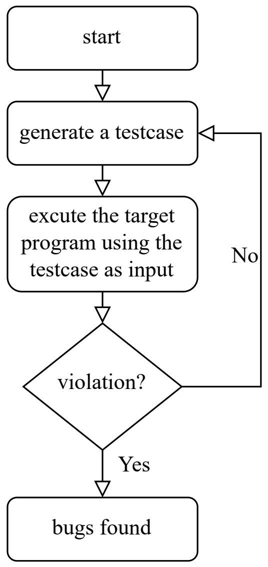


Fig. 1. The working process of fuzzing.

Fig. 1 descripes the main working process of fuzzing, including four stages, the testcase generation stage, testcase running stage, program execution state monitoring and analysis of exceptions. A fuzzer is the automation of this process.

A fuzzing starts from generating a bunch of program inputs, namely, testcases. The quality of testcases directly effects the test results, and determines the quality of the bugs finally found. The inputs should meet the requirement of tested programs for the input format as far as possible. While on the other hand, the inputs should be broken enough so that the program would very likely to fail or crash on these inputs.

Because of the various target program, corresponding inputs would be also various, such as files with different file formats, network communication data, executable binaries with specified characteristics and so on. The main challenge for fuzzing is how to generate such high-quality testcases, which is broken enough. Generally, two kind of generators are used in state-of-the-art fuzzers, generation-based generators and mutation-based generators [5], which will be introduced in the following subsections.

The target program will use the generated testcases as its input. Fuzzers execute the target program automatically, including starting the program with inputs and handle the whole lifetime of a testcase. The analysts of a fuzzer would determin how target programs start and finish, and automatically configure the parameters and the environment variables. Generally, most fuzzers will repeat the loop process until the program execution crashes, which means, a bug is found. But the results are not always optimistic, so the fuzzer stops at a predefined timeout point to handle the situations where no bugs are found.

Fuzzers monitor the execution state during the execution of target programs [4], expecting exception and crashes. Generally, the main monitoring target is to the specific system signals, crashes, and other violations. There are a lot of tools attempting to do this, which is usually called XX sanitizer, meaning checker, such as

- Address Sanitizer [6]
- DataFlow Sanitizer [7]
- Thread Sanitizer [8]
- Leak Sanitizer [9]

After the execution stage finish, the analysts try to determine the location and root cause of captured violations, usually with the help of debuggers, like GDB or other crash analysis tools, like Pin [10], which is another important field of research.

1) Generation-based Fuzzing: Just as its name implies, testcases are generated according to some knowledge in a generation-based fuzzer. That means, for a generation-based fuzzer, knowledge of program input is required, so it is hard to start. For example, for file format fuzzing, usually a configuration file that predefines the file format is provided. Testcases are generated according to the configuration file. With given input knowledge, testcases generated by generation-based fuzzers are able to pass the validation of programs more easily and could be more likely to test the deeper code of target programs.

2) Mutation-based Fuzzing: However, without a friendly document, analyzing and formalizing the input information is a tough work. So the idea of mutation-based fuzzing appears. For a mutation-based fuzzers, only a set of valid initial inputs are needed.

Testcases are generated through the mutation of initial inputs and during the fuzzing process. Moreover, this idea plays an important role in the method of Grammar-based Fuzzing, which will be introduced in the following subsection.

B. Grammar-based Fuzzing

Grammar-based fuzzing (GBF) is a fuzzing technique that uses a formal language grammar to define the structure of the data to be generated. These grammars are typically represented in plain-text and use a combination of symbols and constants to represent the data. The fuzzer can then parse the grammar and use it to generate fuzzed output. Grammar-based fuzzing has been proved to have effective performance in finding bugs and generating good fuzzing files. Compared with other fuzzing methods, grammar-based fuzzing can cost shorter time and less resources to generate fuzzing files. However, different method has its own dvantages and disadvantages.

A grammar-based fuzzing can be guided by machine learning method. In machine learning based fuzzing, a fuzzer must have input files (data set) to learn their grammar format first. Then the fuzzer trains a model to learn the grammar format from the data set. The fuzzer generates a testcase based on the model. In common grammar-based fuzzing, the input grammar is typically written by hand, and this process is laborious, time consuming, and error-prone. With the machine learning method, the testcases are generated automatically. Godefroid et al. designed Learn&Fuzz [11] which is based on recurrent neural network for learning a statistical input model that is also generative which can be used to generate new inputs based on the probability distribution of the learnt model. Learn&Fuzz did experiments on PDF parser. Hu et al. use generative adversarial network to reveal errors and vulnerabilities inside implementations of industrial network protocols [12]. It shows that the machine learning methon has better performance than traditional method.

C. Probabilistic Grammar Fuzzing

Probabilistic grammar fuzzing focus on the problem of generating uncommon (but otherwise syntactically correct and perfectly legal) inputs that are unlikely to be seen in typical operation. So that uncommon inputs would exercise code that is less frequently used in production, possibly less tested, and possibly less well understood. Using such a grammar, we can parse existing common input samples and count how frequently specific elements occur in these samples.Armed with these numbers, we can enrich the grammar to become a probabilistic grammar, in which choices present in productions carry different likelihoods. Since these probabilities come from the common samples used for the quantification, this grammar describes the distribution of valid, but uncommon inputs.

III. GBF: Traditional method

This section, we will give a literarure review on Grammar-based fuzzing using traditional method. Here, the traditional method of GBF is refer to mutation, machine learning (e.g. neural networks), and evolutionary computing (e.g. genetic algorithm and genetic programming), which are widely used in this field. The probabilistic method, which raised by Eberlein et al. [13], will be discussed in the next section.

A. Mutation-based

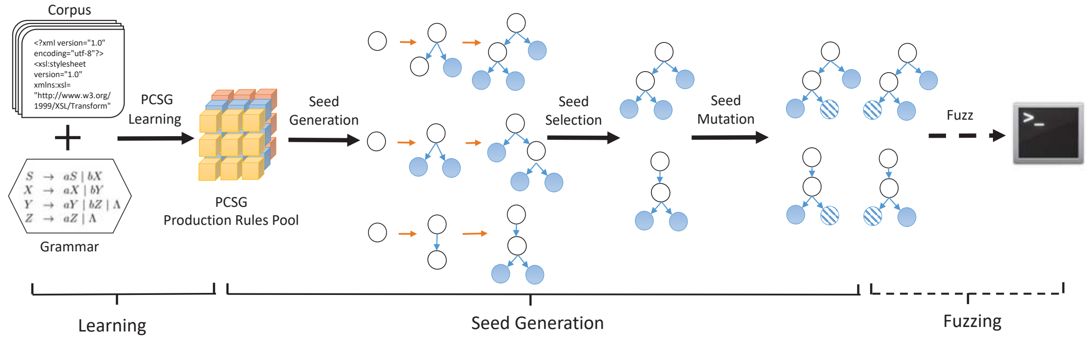
The concept of the mutation technique is collecting data (files, network packets, texts, etc.) then modifying or manipulating them randomly or based on some strategies [14]. Some studies that used mutation technique introduce mutation-based methods to the GBF.

- SD-Gen [15], which is presented by Sargsyan et al., 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, programs are generated based on BNF rules and mutated. SD-Gen supports generating programs in C, C++, Java, Python, etc. Results showed SD-Gen is able to increase code coverage [15]. However, it can't provide programs semantic correctness.
- Saffron [16], which is introduced by Le et al., is an adaptive grammar-based fuzzing approach to effectively and efficiently generate inputs that expose expensive executions in programs. Saffron takes as input a user-provided grammar, which describes the input space of the program under analysis, and uses it to generate test inputs. When given a grammar, Saffron attempts to discover whether the program accepts unexpected inputs, and then repairs the grammar via grammar mutations. Preliminary evaluations showed superior performance of Saffron over a state-of-the-art performance fuzzer on five subject programs from the recent DARPA challenges [16], but it still need more subjects with larger size evalutions.

B. Machine Learning-based

Machine learning is another method that grammar-based fuzzing is using in some available tools. For example, neural network [11] can support learning grammar of inputs from large corpus. Also, neural network can help in generating good inputs are able to increase the code coverage in a target program. The advantage of machine learning approach is generating large diverse test input but better learning does not mean better fuzzing which means the generated better learned input files do not guarantee better fuzzing results.

- Learn&Fuzz [11], designed by Godefroid et al., uses machine learning (Neural Network) 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 [11]. Unfortunately, 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.



Corpus
<?xml version="1.0" encoding="utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl= "http://www.w3.org/1999/XSL/Transform"
$S \to aS \mid bX$
$X \to aX \mid bY$
$Y \to aY \mid bZ \mid \Lambda$
$Z \to aZ \mid \Lambda$
Grammar
PCSG Learning
PCSG Production Rules Pool
Seed Generation
Seed Selection
Seed Mutation
Fuzz
Learning
Seed Generation
Fuzzing

Fig. 2. The Overview of Skyfire

- Skyfire [17] by Wang et al. is a data-driven seed generation approach. It takes a corpus and grammar as inputs and generates inputs in two steps as shown in Fig 2. 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. 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 [17]. However, it is only limited for files that their format are XML, XSL, and JavaScript.

C. *Evolutionary-based*

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 [18] which provide population of individuals' pressure by environment that causes the natural selection (keeping the fittest) which leads to increased fitness of population.

- The fuzzing tool Grammarinator [19], created by Hodován et al., 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 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 [19]. However, it is only fuzzing JavaScript engines.
- IFuzzer [20], a fuzzing tool developed by Veggalam et al., is using evolutionary computation techniques such as genetic programming to guide fuzzing. IFuzzer 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 [20]. Moreover, it found 40 bugs in old version of JavaScript interpreter of Mozila and 17 bugs in latest version of the interpreter [20]. However, IFuzzer is less quality in code generation and for new language or new code IFuzzer needs necessary changes.

IV. GBF: Probabilistic method

Grammar-based fuzzing is a valuable technique for testing the reliability, performance, and security of software. By defining a set of rules that describe the valid structure of the input, it is possible to generate test cases that are more likely to uncover issues, and automate the process of running these tests to save time and resources. Unfortunately, as with any testing method, the traditional grammar-based fuzzy testing we discussed earlier has its limitations, including the fact that it relies on the accuracy and completeness of the rules and may not detect certain types of problems, such as logic errors or race conditions.

So the new GBF application based on probabilistic method has gained the attention of many researchers. Probabilistic grammar-based fuzzing is a variant of grammar-based fuzzing that uses probabilistic techniques to generate test cases. Probabilistic grammar-based fuzzing offers several advantages over traditional grammar-based fuzzing, including greater flexibility, wider range of test cases, and improved efficiency.

In fact, probabilistic grammar-based fuzzing is not a novel concept and has been used by a number of researchers in earlier years. As early as 2011, the Csmith [21] propose a probabilistic context-free grammar (PCFG) to generate C programs. Latter, LangFuzz [22] using the similar PCFG but extend it to more programing language. In addition, some machine learning-based GBFs, such as Learn&Fuzz [11], use the learned input probability distributions to intelligently guide where fuzzy inputs are needed. Recently, Pavese et al. [23] presented an approach to generate test inputs using a grammar and a set of input seeds. By using the input seeds to obtain a probabilistic grammar, Pavese et al. generate similar inputs to the seeds, or by inverting probabilities of the grammar, generate dissimilar inputs.

Overall, probabilistic grammar based fuzzing is a powerful method for testing software that can help uncover a wide range of bugs and other issues. Next we will discuss three applications that use probabilistic method to implement GBF.

A. EvoGFuzz [13]

EvoGFuzz, propesed by Eberlein et al., is a language-independent evolutionary grammar-based fuzzing approach to detect defects and unwanted behavior. The key idea is to combine probabilistic grammar-based fuzzing and evolutionary computing, which aims for directing the probabilistic generation of test inputs toward ”interesting” and ”complex” inputs. The motivation is that ”interesting” and ”complex” inputs more likely reveal defects in a software under test (SUT). For this purpose, the author extend an existing probabilistic grammar-based fuzzer [23] with an evolutionary algorithm.(Fig. 3)

Similarly to the original fuzzer, EvoGFuzz requires a correctly specified grammar for the target language, that is, the input language of the SUT. It can be divided into 8 phases, phase 1-3 is about Probabilistic Grammar-Based Fuzzing from [23] and phase 4-8 is about evolutionary algorithm:

1) in Fig. 3. Create a so-called counting grammar from this grammar that describes the same language but allows us to measure how frequently individual choices of production rules are used when parsing input files of the given language.
2) Learn a probabilistic grammar from a sampled initial corpus of inputs.
3) Use the probabilistic grammar to generate more input files that resemble features of the initial corpus, that is, “more of the same” [23] is produced.
4) Starts a new iteration with analyzing each individual using a fitness function that combines feedback and structural scores.
5) If the stopping criterion is fulfilled (e.g., a time budget has been completely used), the exception-triggering input files are returned. Otherwise, the “interesting” and most “complex” individuals are selected for evolution.
6) The selected individuals are used to learn a new probabilistic grammar, particularly the probability distribution for the production rules similarly to phase 2 that, however, used a sampled initial corpus of inputs.
7) To mitigate a genetic drift toward specific features of the selected individuals, mutate the new probabilistic grammar by altering the probabilities for randomly chosen production rules.
8) Finally, using the mutated probabilistic grammar, we again generate new input files starting the next evolutionary iteration.

Assuming that inputs similar to “interesting”and “complex” inputs more likely reveal defects in the SUT, EvoGFuzz guides the generation of inputs toward “interesting” and “complex”inputs by iteratively generating, evaluating, and selecting such inputs, and learning (updating) and mutating the probabilistic grammar. In contrast to typical evolutionary algorithms, EvoGFuzz does not directly evolve the individuals (test input files) by crossover or mutation but rather the probabilistic grammar whose probabilities are iteratively learned and mutated. In the following, we will discuss each activity of EvoGFuzz in detail.

B. Markov Fuzz [24]

A grammar-based fuzzer can modify the sample input file line by line to produce fuzzing input files, guided by the extracted grammar. However, the biggest limitation of the fuzzer, similar to many existing fuzzers, is that the tool is limited to the quality of the sample input files. In addition, by making modifications on the sample input files, the fuzzer sticks with the order of commands in the given sample input files, hence it is greatly limited by the contents of the provided sample input files. This situation makes the tool frequently visits the same locations or goes through the same paths in the code, which hinders the tool from going to other paths or locations in the code that could find a vulnerability.

Learning the order of the commands and generating new fuzzing input files without modifying the sample input files will be our target. To achieve that, the Markov chain model can help generate new fuzzing input files.

The fuzzer applies the Markov chain model and analyzes and study the sample input files to collect information about the orders and probabilities of each command. There are three major stages: (1) Analyzing commands and parameters by learning the command order by using Markov chain model, (2) Calculating command and parameter probabilities, and (3) Generating completely new fuzzing input files with all of the information learned from analyzing sample input files.

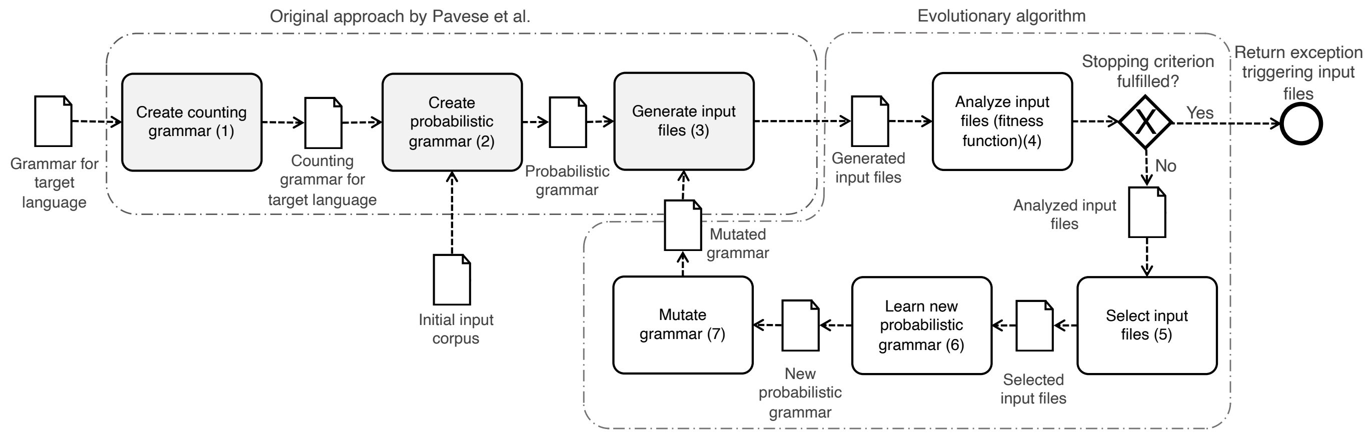

Original approach by Pavese et al.
Evolutionary algorithm
Grammar for target language
Create counting grammar (1)
Counting grammar for target language
Create probabilistic grammar (2)
Probabilistic grammar
Generate input files (3)
Initial input corpus
Generated input files
Analyze input files (fitness function)(4)
Stopping criterion fulfilled?
Yes
No
Return exception triggering input files
Analyzed input files
Select input files (5)
Selected input files
Learn new probabilistic grammar (6)
New probabilistic grammar
Mutate grammar (7)
Mutated grammar

Fig. 3. The Overview of EvoGFuzz

1) Analyzing Commands and Parameters Using Markov Chain Model: Firstly, we entail extracting commands from sample input files without sorting so the fuzzer can keep order of occurrence in sample input files. Then we add a start mark and an end mark in the command list. Secondly, we should calculate commands transition probablities. When implementing the Markov chain model in our fuzzer, the calculated probability of each next command for a current state command is saved in two groups. So, the current state command has one or more commands following it with their probabilities. When a command is found, the tool chooses the next command from the group based on the probability of that command. The tool repeats these steps for upcoming commands until it stops. Then we need to find out how many parameter matches there are for each command in the grammar and the probabilities of parameters. Finally, the fuzzer calculates the occurrences of each command in the sample input files. Then, it divides the parameters' total count numbers by the calculated number to get each parameter' s probability. By knowing this, the fuzzer knows how to choose a parameter for a command based on the probability of parameters.

2) Generating New Input Lines: The fuzzer uses list of commands and parameters probabilities to generate new input lines. Taking each list of commands, the tool picks parameters for each command in the list. If the command in the list matches a command in the grammar, the fuzzer chooses the parameter for that command based on the probability. Then it appends the selected parameter to the command and saves it as an input line. The tool repeats this for upcoming commands in the list. The input lines will be saved to tell the fuzzer how to generate a fuzzing file.

3) Generating Completely New Fuzzing Input Files: After getting the input format files from previous steps, the tool will use them to generate new fuzzing files. it takes one of the input format files and changes only one line and keeps other lines unchanged. This stage will be repeated many times until the tool generates enough new fuzzing input files.

C. TreeFuzz [25]

For input data that can be represented as a labeled, ordered tree, which covers many common formats, such as source code (represented as an AST), documents (PDF, ODF, HTML), and images (SVG, JPG), TreeFuzz learns models of such input data by traversing each tree once while accumulating information. For each node and edge in the tree, TreeFuzz gathers facts that explain why the node or edge has a particular label and appears at a particular position in the tree. After having traversed all input data, the approach summarizes the gathered information into probabilistic models. Finally, based on the learned models, TreeFuzz generates new input data by creating trees in a depth-first manner.

However, as a language-independent, blackbox fuzz testing approach, TreeFuzz still can generate input data that mostly complies with the expected input format, and enables testing a variety of programs that expect structured input data, and is efficient and effective at finding browser inconsistencies.

TreeFuzz consists of three phases. First, during the learning phase, the approach infers from a corpus of examples a set of probabilistic, generative models that encode properties of the input format. Second, during the generation phase, TreeFuzz creates new data based on the inferred models. Finally, the generated data serves as input for the fuzz testing phase.

1) Learning:

```
var valid = true , val = 0;
if ( valid ) {
    function foo (num) {
        num = num + 1;
        valid = false ;
        return ;
    }
    foo ( val );
}
```

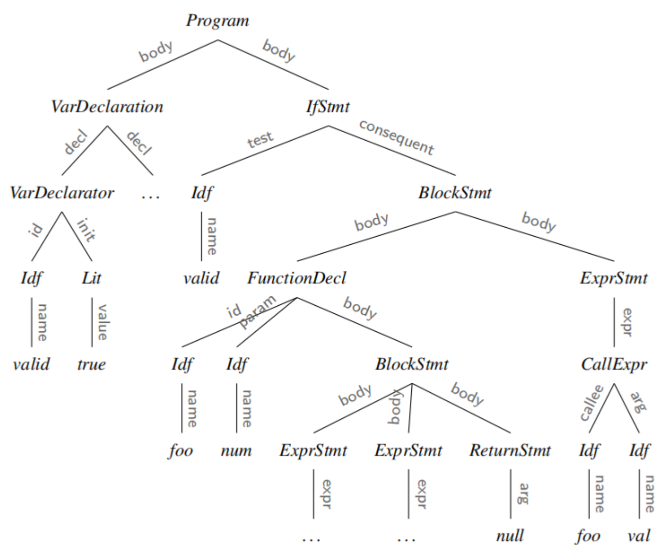


Fig. 4. TreeFuzz of the example

In the example under JavaScript language, TreeFuzz traverses the tree of the example (Fig. 4) while inferring probabilistic, generative models of the input format, and capture structural properties of the tree. For example, as some basic properties, the approach infers that nodes labeled *Program* have outgoing *body* edges and that these edges may lead to nodes labeled *VarDeclaration* and *IfStmt*. Additionally, the approach infers the probability of particular destination nodes, like nodes labeled *BlockStmt*, an outgoing edge *body* leads to an *ExprStmt* three out of five times. Furthermore, TreeFuzz infers more complex properties in addition to PCFG(probablity CFG)-like properties. For example, since nodes labeled *ReturnStmt* always occur as descendants of a node *FunctionDecl*, the approach infers that return statements occur inside functions. What's more, the *id* edge of node *FunctionDecl* and the callee edge of node *CallExpr* lead to identical subtrees, so it infers *FunctionDecl* and *CallExpr* have a definition-use-like relation.

2) Generation: After analyzing the given tree, TreeFuzz generates new trees, based on the inferred models. For example, to enforce the constraint that return statements must appear within a function declaration, TreeFuzz only creates a *ReturnStmt* node when the currently expanded node is a descendant of a *FunctionDecl* node. Consequently, the return statements must be within a function. What's more, suppose the approach generates the *Idf* subtree of a *CallExpr* node. To select a label for the destination node of an edge *name*, the approach checks whether there already exists a *FunctionDecl* node with a matching subtree, and if so, reuses the label of this subtree. Consequently, most generated function calls have a corresponding function declaration, and vice versa.

In this simple example, the model that TreeFuzz infers overfits the example, so the generated programs do not use all features of the JavaScript language. So if the corpus of examples is large enough, the model is general enough to create a variety of other valid examples that go beyond the corpus.

3) Testing: Finally, the data generated by TreeFuzz is given as input to programs under test. For the running example, consider executing the generated programs in multiple browsers to compare their behavior. Executing the following program will exposes an inconsistency between Firefox 45 and Chrome 50.

```
var val = true , valid = true ;
if ( val ) {
    foo ( val );
    function foo (num) {
        return ;
        return ;
        val = num + 1;
    }
}
```

V. Comparison

TABLE I
Compare the GBF property between traditional and probabilistic methods

property	traditional	probabilistic
Hardness	medium	easy
Speed	low	extremely low
Input Validity	good	perfect
Code coverage	normal	high
Efficiency	normal	high
Flexibility	low	high

In this section, we compare the property of traditional and probabilistic methods.

Our comparison focuses on the following research questions, and we answer four of them in the following subsections:
RQ1 Hardness: How much effort is it to set up a binary template file for input generation?
RQ2 Speed: How fast is a fuzzer to produce inputs?
RQ3 Input Validity: How accurate is a fuzzer in producing inputs?
RQ4 Code coverage: Does the inputs generated traverse as many program running states as possible?
RQ5 Efficiency: How many bugs can a fuzzer found?
RQ6 Flexibility: How widely can a fuzzer be used?

A. Hardness

For a generation based fuzzer, knowledge of program input is required. Correspondingly, a grammar-based fuzzer need a definition of context-free grammar, which should be written by researcher or automatically generated by fuzzer. In both cases, the priori knowlege is unavoidable, which is hard.

However, the probabilistic methods randomly generate the context-free grammar by the process of ”learning”, and do not need too much priori knowledge, which is relatively easy to start.

TABLE II
Exception types that have been triggered by both approaches.

Input	Subject	Exception	EvoGFuzz	Baseline
JSON	ARGO	argo.saj.InvalidSyntaxException	30 / 30	0 / 30
	Genson	java.lang.NullPointerException	30 / 30	30 / 30
	jsonToJava	org.json.JSONException	30 / 30	30 / 30
	jsonToJava	java.lang.IllegalArgumentException	30 / 30	30 / 30
	jsonToJava	java.lang.ArrayIndexOutOfBoundsException	30 / 30	30 / 30
	jsonToJava	java.lang.NumberFormatException	6/ 30	0 / 30
	Pojo	java.lang.StringIndexOutOfBoundsException	30 / 30	30 / 30
	Pojo	java.lang.IllegalArgumentException	30 / 30	30 / 30
	Pojo	java.lang.NumberFormatException	22 / 30	0 / 30
JavaScript	Rhino	java.lang.IllegalStateException	26 / 30	0/ 30
	Rhino	java.util.concurrent.TimeoutException	15 / 30	0 / 30
CSS3	No exceptions triggered			
Total exception types			11	6

TABLE III
Input Validity of TreeFuzz

target language	validity type	validity rate
JS	syntactically valid	96.3%
	w/o runtime error	14.4%
HTML	W3C markup validator [26]	2.06 errors/kB

B. Efficiency

To answer RQ5, we compare the number of times a unique exception type has been triggered of EvoGFuzz. Table. II shows the thrown exception types per subject and input language. If neither approach was able to trigger an exception, the subject is not included in the table. For the Gson, JsonJava, simple-json, minimal-json, and cssValidator parsers no defects and exceptions have been found by both approaches. The ratios in the 4th and 5th column relate to the number of experiment repetitions in which EvoGFuzz and the baseline were able to trigger the corresponding exception type.

Table. II show that during each experiment repetition, EvoGFuzz has been able to detect the same exception types than the baseline. Furthermore, EvoGFuzz was able to find five additional exception types that have not been triggered by the baseline.

C. Input Validity

Take TreeFuzz as an example in Table. III. TreeFuzz generates trees that are intended to comply with an input format without any a priori knowledge about this format. To assess how effective the approach is in achieving this goal, we measure the percentage of generated trees that pass language-specific validity checks.

1) JavaScript: To measure whether a generated JavaScript program is valid, we pretty print it and parse it again. If the pretty printer rejects the tree or if the parser rejects the generated program, then we consider the program as syntactically invalid. 96.3 % of 100,000 generated trees represent syntactically valid JavaScript programs. Furthermore, 14.4% of the syntactically valid programs execute without causing any runtime error.

2) HTML: To measure the validity of generated HTML documents, we use the W3C markup validator [26]. In practice, most HTML pages are not fully compatible with the W3C standards and therefore cause validation errors. As a measure of how valid an HTML document is, we compute the number of validation errors per kilobyte of HTML.

The generated HTML documents have 2.06 validation errors per kilobyte. As a point of reference, the corpus documents contain 0.59 validation errors per kilobyte. That is, the generated documents have a slightly higher number of errors, but overall, represent mostly valid HTML. We conclude that TreeFuzz effectively generates HTML documents that mostly comply with W3C standards, without any a priori knowledge of HTML.

To the best of our knowledge, there is no existing approach based on a learned, probabilistic language models that generates entire programs with so few mistakes.

D. Code Coverage

Eberlein et al. [13] examine the effectiveness of EvoGFuzz and evaluate the approach on the same test subjects that Pavese et [23] have originally covered with their proposed probabilistic grammar-based fuzzing approach. These test subjects require three, in complexity varying input formats, namely JSON, JavaScript, and CSS3. ARGO, Genson, Gson, JSONJava, JsonToJava, MinimalJson, Pojo, and json-simple serve as the JSON parsers, whereas Rhino and cssValidator serve as the JavaScript and CSS parser, respectively. summarize the conclusion the code coverage performance of EvoGFuzz and Baseline. From the Table. IV, we can see that EvoGFuzz improves the coverage for all subjects. For both approaches, Table. IV shows the maximum and median line coverage, the standard deviation as well as the number of generated input files, along with the increase of the median line coverage of EvoGFuzz compared to the baseline. The improvement of the median line coverage ranges from 1.00% (Pojo) to 47.93% (Rhino). Additionally, the standard deviation (SD) values for the baseline in Table. IV indicate the existence of plateaus because all repetitions for each JSON parser show a very low (and often 0%) SD value.

VI. Conclusion

In conclusion, the paper has discussed the evolution of grammar-based fuzzing from traditional to probabilistic

TABLE IV
Coverage Results for Each subject

Subject	EvoGFuzz			Baseline			Median increase
	max	median	SD	max	median	SD	
ARGO	49.78%	48.48%	0.60%	43.19%	43.19%	0%	12.25%
Genson	19.65%	19.09%	0.19%	16.17%	16.17%	0%	18.06%
Gson	26.92%	26.67%	0.15%	24.08%	24.08%	0%	10.76%
JSONJava	21.09%	18.47%	0.59%	16.72%	16.72%	0%	10.47%
JsonToJava	18.58%	17.90%	0.39%	17.58%	17.45%	0.09%	2.58%
minimalJson	51.06%	50.83%	0.26%	46.38%	46.38%	0%	9.59%
Pojo	32.33%	32.17%	0.07%	31.88%	31.88%	0.02%	1.00%
json-simple	34.44%	33.80%	0.33%	28.54%	28.54%	0%	18.43%
Rhino	15.42%	13.95%	0.43%	10.20%	9.43%	0.28%	47.93%
ccsValidator	7.53%	7.06%	0.21%	6.62%	6.51%	0.06%	8.45%

methods. It has shown that while traditional grammar-based fuzzing relies on pre-defined grammar rules to generate test inputs, probabilistic methods use machine learning techniques to learn the grammar of the target program and generate more diverse and effective test inputs. The effectiveness of probabilistic methods has been demonstrated through experiments on several real-world programs. Overall, the use of probabilistic grammar-based fuzzing can greatly improve the effectiveness of testing and help identify a wider range of potential bugs in software.

VII. Futrue Work

Future work on probabilistic grammar-based fuzzing could focus on several areas, including the following topic:

1) Effectiveness: Future research could focus on developing new probabilistic algorithms and techniques for generating inputs that are more likely to uncover bugs and vulnerabilities in software systems. This could include the use of machine learning algorithms to automatically learn grammars from training data, or the incorporation of feedback from test execution to guide input generation.

2) Debugging: Mined probabilistic grammars could be used to characterize the features of failure-inducing inputs, separating them from those of passing inputs. Statistical fault localization techniques [27], for instance, could then identify input elements most likely associated with a failure. Generating "more of the same" inputs, and testing whether they cause failures, could further strengthen correlations between input patterns and failures, as well as narrow down the circumstances under which the failure occurs.

3) Application: Probabilistic grammar-based fuzzing is a new research direction, and there are still relatively few studies related to it. Future research can evaluate the use of probabilistic grammar-based fuzzing in different domains, such as cybersecurity, mobile app testing, and the internet of things, to determine its effectiveness and potential challenges in these domains.

References

[1] Z.-y. Wu, H. Wang, L. Sun, Z. Pan, and J. Liu, "Survey of fuzzing," *Application Research of Computers*, vol. 27, no. 3, pp. 829–832, 2010.

[2] H. Al Salem and J. Song, "Grammar-based fuzzing tool using markov chain model to generate new fuzzing inputs," in *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2021, pp. 1924–1930.

[3] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[4] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[5] I. Van Sprundel, "Fuzzing: Breaking software in an automated fashion," *Decmember 8th*, 2005.

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[7] DataFlowSanitizer. [Online]. Available: {https://clang.llvm.org/docs/DataFlowSanitizer.html}

[8] ThreadSanitizer. [Online]. Available: {https://clang.llvm.org/docs/ThreadSanitizer.html}

[9] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

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

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

[13] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, "Evolutionary grammar-based fuzzing," in *International Symposium on Search Based Software Engineering*. Springer, 2020, pp. 105–120.

[14] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

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

[16] X.-B. D. Le, C. Pasareanu, R. Padhye, D. Lo, W. Visser, and K. Sen, "Saffron: Adaptive grammar-based fuzzing for worst-case analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 44, no. 4, pp. 14–14, 2021.

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

[18] C. Darwin, *On the origin of species, 1859*. Routledge, 2004.

[19] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, 2018, pp. 45–48.

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

[21] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[22] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.

[23] E. Pavese, E. Soremekun, N. Havrikov, L. Grunske, and A. Zeller, “Inputs from hell: generating uncommon inputs from common samples,” *arXiv preprint arXiv:1812.07525*, 2018.

[24] H. Al Salem and J. Song, “Grammar-based fuzzing tool using markov chain model to generate new fuzzing inputs,” in *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2021, pp. 1924–1930.

[25] J. Patra and M. Pradel, “Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data,” *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[26] W3C markup validation service. [Online]. Available: {https://validator.w3.org/}

[27] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.