Mathematical Problems of Computer Science 50, 67--75, 2018.

Feedback Driven Grammar-Based Fuzzing

Seryozha A. Asryan

Institute for Informatics and Automation Problems of NAS RA
e-mail: asryan@ispras.ru

Abstract

In this paper, we present a method for grammar-based fuzzing, which improves its penetration power. It is based on input data generation using a fuzzer feedback. Several other methods are prone to create an initial set of acceptable test cases before the actual fuzzing process, and hence are unable to use the runtime information to increase the generated input's quality. The proposed method uses the coverage information gathered for each input sample and guides grammar-based input generation. This method uses more than 120 BNF (*Backus-Naur Form*) grammar rules described in ANTLR (*Another Tool for Language Recognition*) platform. Experimental results show that our method - feedback driven random test generation, has higher code coverage capabilities compared with the existing methods.

Keywords: Fuzzing, BNF grammars, Structured data, Automated test generation

1. Introduction

Correctness of compilers is crucial to most software projects. Random testing, or fuzzing, has emerged as an important tool for finding bugs in compilers and runtimes. The main idea behind fuzzing is to feed target application (*program under test*) with a large amount of mutated test inputs to trigger unintended program behavior like hangs and crashes.

The existing fuzzing approaches can be classified in three basic categories – blackbox fuzzing, whitebox fuzzing and graybox fuzzing. Blackbox fuzzers had no knowledge about the program's internal structure and, hence, are less effective. Sometimes they can use grammars to generate inputs with specific characteristics. The second type of fuzzers usually combine fuzzing with heavy-weighted symbolic execution to improve the effectiveness by applying a symbolic engine in cases where fuzzer is unable to explore a new execution path (*i.e., increase the code coverage*). Graybox fuzzing is something in between. It uses a light-weighted program instrumentation to extract partial information of the program to generate guided input samples without sacrificing execution speed.

Unfortunately, when it comes to testing applications with complex structured-inputs, fuzzers have several limitations. Examples of such applications can be compilers, translators and interpreters. These applications have a multi-pass design and process input in multiple stages

(lexer, parser etc.). Because of complicated checks and a huge amount of possible execution paths at the first stage, fuzzers are mostly unable to generate inputs that could exercise code beyond the first stage.

Currently there are several methods and instruments [1-6, 8-11] that use whitebox fuzzing or grammar-based data generation approaches to address the challenge. CSmith [1] has been successfully used to identify hundreds of bugs in C/C++ compilers, however this and other similar approaches have significant drawbacks. CSmith couples input generation logic with target programming language specifications. Producing inputs based on this strategy requires expert knowledge and a significant engineering effort, which needs to be repeated from scratch for each new language. For example, to support a new programming language, this method requires the definition of a corresponding grammar and manual implementation of language features. Grammar-based whitebox fuzzing [2] enhances the whitebox fuzzing technique by using the input grammar specification to construct valid test cases. It presents a dynamic test generation algorithm that uses symbolic execution to directly generate grammar-based constraints, the satisfiability of which is checked using a custom grammar-based constraint solver. The two main disadvantages of this method are the long runtime and the small set of available grammars. Another instrument SynTESK (Syntax Testing Kit) [3] implements UniTESK [4] technology. Using BNF grammars, SynTESK generates two sets of programs. The first set contains test cases, which will be accepted by compiler. In the second set instruments collect invalid programs, which will be rejected by the compiler. GramFuzz [5] can automatically detect grammar rules based on the provided input samples. After that, those grammars are used for further data generation. This instrument is mainly used for web browsers fuzzing. In its first step GramFuzz considers a set of inputs of HTML, CSS and JavaScript. Then it tries to extract BNF rules and construct corresponding AST (*Abstract syntax Tree*). During fuzzing, inputs are generated by replacing the AST nodes with available elements (duplications are also accepted). The main limitations of this instrument are: not all BNF rules can be extracted from the provided samples, not all generated inputs are valid programs for parser. Another instrument Ifuzzer [6] finds bugs in JavaScript interpreters. It uses evolutionary computing techniques, such as genetic algorithms [7], to guide the fuzzer in generating uncommon input code fragments that may trigger exceptional behavior in the interpreter. Ifuzzer gets as an input the context-free grammar of a particular language and a test suite with valid inputs. Based on that grammar, it generates parse trees and extracts code fragments (*fragment pool*) from a given test-suite. The initial population for a genetic algorithm consists of random selection of programs, from the input test samples. For each new generation, Ifuzzer uses the fitness function on inputs from the previous generation to determine a set of inputs upon which the new generation should be constructed. Elements of the next generation are created by selecting random code fragments of the appropriate input code for replacement. Replacement was performed by choosing a random member from the fragment pool. Fitness function consists of a fuzzer feedback (whether the program crashed or not) and input complexity. This method uses only the final state of program execution as feedback and doesn't consider information about the program execution paths (*code coverage information*), as well as the correlation between program paths. Instrument Superion [8] proposes a grammar-aware coverage-based grey-box fuzzing. This instrument uses the grammar of its test inputs to parse each input into an AST. Based on the constructed AST, Superion performs a trimming operation to reduce the size of the inputs, iteratively removing each subtree in the AST of a test input and observing coverage differences. It uses also two types of mutation strategies. The first one is AFL's [9] standart dictionary mutation. The second one replaces one subtree in the AST of a test input with the subtree from itself or another test input in the queue. One of the limitations of Superion is that it needs well-documented grammars, as well as an initial set of valid test cases. The paper [10] proposes the instrument BlendFuzz that uses grammars to create syntactically valid test inputs and guide test generation. This approach

consists of two stages. The first stage requires an initial set of valid inputs and a corresponding grammar. According to the provided grammar, BlendFuzz breaks test cases into grammatical fragments, which will be used as basic building blocks in the next stage. In the second one, generated code fragments are used to mutate the existing test cases. More specifically, BlendFuzz selects a grammatical fragment of one input and replaces it with another one in the pool. This procedure is repeated systematically to generate a large set of input samples. Although BlendFuzz is quite successful in practice, it is based on random testing technique and doesn't incorporate results of program execution. The paper [11] describes a learn&fuzz algorithm that uses sample inputs and neural-network-based statistical machine-learning techniques to automatically generate input grammars for grammar-based fuzzing. This algorithm can also generate new inputs based on the probability distribution of the learnt model. Learn&fuzz algorithm is trained over a corpus of PDF files to generate test inputs for the Microsoft Edge PDF parser. The results can vary depending on different input formats and training sets.

After studying the variety of methods mentioned above, we came to the conclusion that these methods have several limitations:

- Input generation strategy requires expert knowledge and a significant engineering effort, which needs to be repeated from scratch for each new language [1]
- Input generation is based only on the usage of the final state of the program execution (whether the program crashed or not) and doesn't consider information about the program execution paths (code coverage information), as well as correlation between program paths [6]
- Method needs well-documented grammars and an initial set of valid test cases [8]
- Test generation is based on random testing technique and doesn't use feedback of program execution [10].

In this paper, we propose a method for generating input data based on target BNF grammars. We develop this method on top of our previous paper [12], which increases the overall effectiveness and percentage of correctly generated input samples that will successfully pass the parsing stage.

Our work makes the following contributions:

- We use the push down automata representation of BNF rules and fuzzer edge coverage information to direct the input generation towards increasing coverage.
- Using the coverage information, we add weighted values to edges (*transitions*) of push down automata of each input sample. Then we use that information at the input generation stage to produce inputs with higher chances to explore new paths of program under test.

The rest of this paper is organized as follows: Section 2 describes our approach to grammar-based data generation. Section 3 provides a detailed description of the implemented model of interaction between a fuzzer and an input generation component (Sd-Gen – Structured Data Generation). Section 4 presents the results of the performed experiments, and comparison with other methods. Finally, Section 5 presents our conclusion.

2. Feedback-Driven Data Generation

2.1 ANTLR's Grammar Structure

ANTLR platform provides BNF grammars for more than 120 different languages. As we discussed in our previous paper [12], each of these grammars has its own set of pushdown automata representations.

```
object: ‘{‘ pair ( ‘,‘ pair ) * ‘}‘
      | ‘{‘ ‘}‘
      ;
 pair:  STRING ‘:‘ value
      ;
```

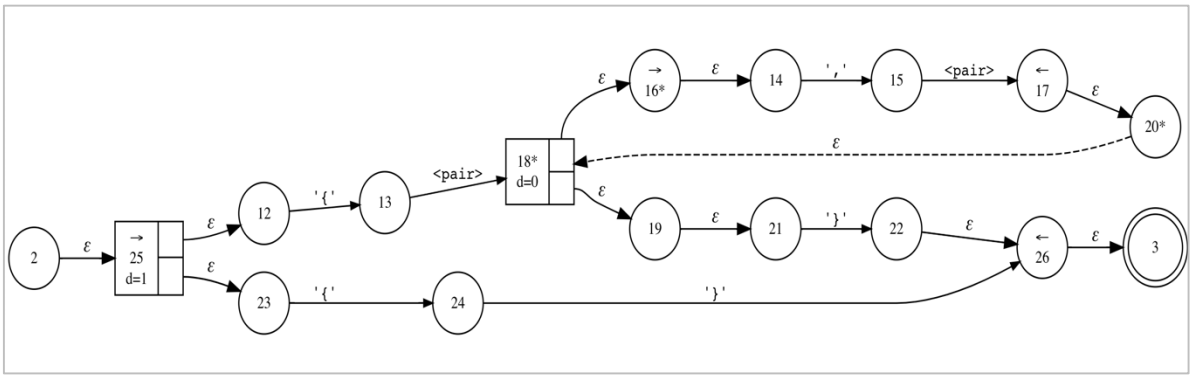

Fig. 1. Grammar rule representation in ANTLR.

Figure 1 shows an example of a rule from BNF grammar and its corresponding pushdown automata. The rule is described with its name (*rule name*), followed by a single alternative, terminated with a semicolon, or it can also have alternatives. Alternatives are either a list of rule elements or an empty list. In the picture above, “*object*” is a rule, which has “*pair*”, “ *‘{’ ‘}’* ” as its alternatives. In ANTLR platform, each rule should have a pushdown automata representation. For “*object*” rule (1) ANTLR would generate pushdown automata (2). We use this representation to distinguish *interesting* rules, which were used to build inputs to exercise new paths in target program during its fuzzing.

2.2 Guided Test Case Generation

It is difficult to generate test cases for compilers, because their inputs are highly structured. Vast majority of existing compiler fuzzing systems generate a set of inputs before actually starting the testing process. Hence, they are not able to change their data generation models based on runtime information. Despite the fact that some instruments use BNF grammars, without instrumentation feedback from the testing system, the generated samples will cover random parts of the target program. Our test generation system tries to overcome these problems by creating dependencies between the generated input structure and information based on the target code coverage.

The method used in our previous version of Sd-Gen (paper [12]) to generate data was based on the following algorithm. It takes two main parameters as input – *depth* ***(D)*** for each rule (that is, the maximum number of recursive selection); *length* ***(L)*** of generated input. Additionally, we use two counters ***cD=0*** and ***cL=0*** to store the current *depth* and *length* values.

1. Select a random rule from the BNF grammar and mark it as a *start* rule. Add that rule to ***rList***. Set the value of ***cL*** to the number of nodes in automata of *start* rule. Go to step 2.
2. Walk through ***rList*** and select the first non-terminal symbol (***NT*** node). If there is no ***NT*** node go to step 6, otherwise continue to step 3.

3. On the corresponding automata of ***NT,*** the rule algorithm selects a random path using the depth first search algorithm (*DFS*). Then it replaces the ***NT*** rule in ***rList*** with the constructed path and updates ***cL*** and ***cD*** values.
4. If ***cL*** and ***cD*** values reached their limits (***cL*** < ***L*** and ***cD*** < ***D*** values), go to step 5, if not, go to step 2.
5. For each non-terminal node in ***rList,*** the algorithm constructs the shortest terminal path and replaces it in ***rList***.
6. Store the content of ***rList*** in output file.

To implement guided input generation, we added weight values to each transition of existing automata (Figure 2).

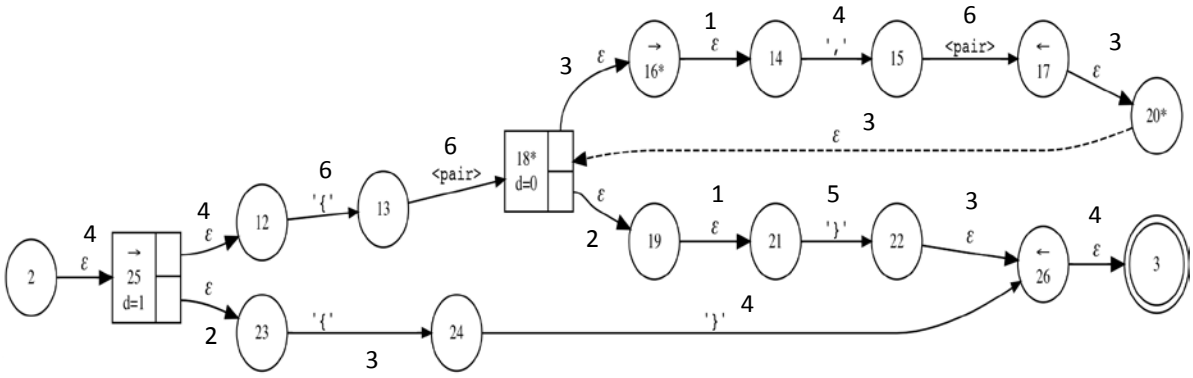

Fig. 2. Automata of BNF rule with weight values.

The above figure shows the automata for "*object*" rule of some BNF grammar, with weights on its edges. Due to those weights we are able to make modification in our algorithm:

- Instead of choosing a random path on the corresponding automata of ***NT*** rule in step 3, now we select a path with the highest weight value.

This can be achieved by choosing each transition using its weight as measurement for probability, i.e., select "*25→12*" rather than "*25→23*" because the weight of the second edge is less than the weight of the first one. To prevent the algorithm from selecting the same transition on every iteration, we decrease its weight by the appropriate value.

3. Implementing Sd-Gen as Plugin for Fuzzing Tool

The current implementation of Sd-Gen as a fuzzer mutation plugin supports the configuration file that can be used to manually specify the number of testcases to be generated and its iteration counts. The BNF data generation plugin is designed to manage the execution of Sd-Gen instrument based on the assigned parameters. This plugin has the following main responsibilities:

- Run Sd-Gen instrument, which generates a set of BNF-based inputs using information about the transition weights received from the fuzzer. If such information does not yet exist, a set of completely random inputs is generated
- For every constructed input, the plugin sends its rule transition information (*including weights, if they exist*) to the fuzzer engine.

On the other hand, after detecting an interesting input, the fuzzer retrieves its rule transition information and updates it based on the target execution coverage results. Each time when Sd-Gen is called, it checks whether there is information about transitions weights or not. If the

required information is available, it will choose the rules and transitions of that rule based on the exited data. Doing this iteratively, improves the generated data accuracy and makes it possible to change the test case generation strategies in runtime, hence significantly increasing the impact of the constructed inputs on the target code coverage.

The BNF data generation plugin is called in two different phases. First, it acts as a mutation plugin to continuously generate new inputs, and second, it is activated whenever fuzzing detects an interesting input (increases coverage or finds some crashes/hangs) sample to update the corresponding weight values.

4. Results

This section provides comparison results of the proposed method with our previous method and also with the existing methods. For analysis purpose we use several well-known compilers and interpreters. One of the main parameters of measurement is the application code coverage, as well as the number of generated inputs to gain that coverage. As shown in Table 1, almost in all cases, our method was able to achieve more code coverage with less inputs. In case of gcc/g++, CSmith gets more coverage results due to its way of generating inputs sample, which, starting from the initial sample, are all valid program instances.

Application name	***Our previous method***	***Feedback driven grammar-based fuzzing (our current method)***	***Execution count***	***Coverage info. (%)***
Gcc-7.1	24959	26107	~9800	**+4.6**
G++-7.1	26154	30103	~8400	**+15.1**
Python-2.7	7561	7962	~41000	**+5.3**
Php-v7.1.7	2036	2107	~325000	**+3.5**
Luac-5.3.4	13395	16274	~9700	**+21.5**
Gfortran-7.1	24060	24950	~5300	**+3.7**

Table. 1. Experimental results for testing of the proposed method

5. Conclusion

Fuzzing is a powerful technique for testing an application by randomly mutating its input values. However, for the certain type of application (compiler, interpreters) it is hard to generate test cases that will not fail at first levels of execution.

Our proposed method implements guided data generation using BNF grammars. We are able to generate new input testcases based on target application feedback on each input sample. It allows us to make modifications in our testcase generation strategies while continuing the fuzzing process. This method results in improvement of target code coverage and increases the analyses effectiveness.

References

[1] Xuejun Yang, Yang Chen, Eric Eide and John Regehr**,** “Finding and understanding bugs in C compilers”, *PLDI '11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Jose, California, USA, pp. 283-294, 2011.

[2] P. Godefroid, M. Y. Levin and D. Molnar, "Grammar-based whitebox fuzzing", *PLDI '08 Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, pp. 206-215, 2008.

[3] С. В. Зеленов и Н. В. Пакулин, “Верификация компиляторов – систематический подход”, *Proceedings of the Institute for System Programming*, Russia, vol. 13, no. 1. стр. 47-64, 2007.

[4] I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin and A. K. Petrenko, "UniTesK Test Suite", *FME '02 Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, London, UK, pp. 77-88, 2002.

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

[6] S. Veggalam, S. Rawat, I. Haller and H. Bos, “Ifuzzer: An evolutionary interpreter fuzzer using genetic programming”, *Proceedings of Computer Security - 21st European Symposium on Research in Computer Security*, pp. 581-601, Sep. 2016

[7] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan and M. O’Neill, “Grammar-based genetic programming: a survey,” *Genetic Programming and Evolvable Machines*, vol. 11, pp. 365–396, May 2010.

[8] J. Wang, B. Chen, L. Wei and Y. Liu, “Superion: Grammar-Aware Greybox Fuzzing”, 2018.

[9] M. Zalewski, web-page - American Fuzzy Lop

[10] D. Yang, Y. Zhang and Q. Liu, “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, London, June 2012.

[11] P. Godefroid, H. Peleg and R. Singh, “Learn&Fuzz: machine learning for input fuzzing”, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 50-59, USA, Nov. 2017

[12] S. Sargsyan, Sh. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan and S. Asryan, “Grammar-based Fuzzing", *Proceedings of Ivannikov Memorial Workshop*, pp. 32-36, Armenia, May 2018.

Submitted 05.08.2018, accepted 04. 12.2018.

Ֆազինգի մեթոդ՝ կոնտեքստից ազատ քերականությունների օգտագործմամբ. տվյալների գեներացիա ֆազերի հետ հետադարձ կապի միջոցով

Ս. Ասրյան

Ամփոփում

Մեր օրերում ֆազինգը համարվում է ավտոմատ թեստավորման ամենաարդյունավետ և ամենահայտնի մեթոդներից մեկը։ Սակայն գոյություն ունեցող մեթոդներն ունեն սահմանափակ հնարավորություններ հետազոտելու այնպիսի ծրագրային համակարգեր (օր.՝ կոմպիլյատորներ), որոնք մշակում են բարդ կառուցվածք ունեցող տվյալներ։

Հոդվածում ներկայացված է ֆազինգի մեթոդ, որն օգտագործում է կոնտեքստից ազատ քերականությունները մուտքային տվյալների կառուցման համար՝ հիմնվելով ֆազինգի ընթացքում ստացված տվյալների վրա։ Գոյություն ունեցող մոթոդներից շատերը հակված են կառուցել տվյալների ամբողջական հավաքածուներ, մինչ ֆազինգի սկիզբը, որի հետևանքով անհնար է դառնում օգտագործել կատարման ընթացքում ստացված ինֆորմացիան կառուցվող տվյալների որակը բարձրացնելու համար։ Առաջարկվող մեթոդի նպատակն է ուղղորդել քերականության վրա հիմնված մուտքային տվյալների կառուցումը՝ օգտագործելով ինֆորմացիա ուսումնասիրվող ծրագրային համակարգի կոդի ծածկույթի մասին։ Այս մեթոդն օգտագործում է ավելի քան 120 քերականությունների BNF (Bakus-Naur Form) ներկայացումներ, նկարագրված ANTLR ծրագրային համակարգում։ Փորձերի արդյունքները ցույց են տալիս, որ ներկայացված մեթոդի միջոցով կառուցված տվյալների շնորհիվ հնարավոր է ստանալ հետազոտվող ծրագրային ապահովման կոդի ավելի մեծ ծածկույթ, քան մյուս հայտնի մեթոդները։

Фаззинг с использованием грамматических правил: генерация данных на основе обратной связи с фаззером

С. Асрян

Аннотация

В наши дни фаззинг является одним из наиболее эффективных и широко используемых методов автоматического динамического анализа. Несмотря на это, существующие методы имеют ограничения при тестировании приложений (компиляторы и интерпретаторы) обрабатывающие входные данные, имеющие сложную структуру.

В статье представлен метод фаззинга на основе грамматических правил. Метод основан на генерации входных данных программы, используя обратную связь с фаззером (информацию в результате выполнения фаззинга). Множество других методов склонны создавать начальный набор тестовых примеров до начала процесса фаззинга, и, следовательно, не могут использовать информацию, доступную во время выполнения фаззинга, для повышения качества генерируемых тестовых примеров. Предлагаемый метод использует покрытие кода программы, собранный для каждого тестового примера и направляет процесс построения новых входных данных (на основе грамматик). Данный метод использует БНФ (Форма Бэкуса — Наура) представления более 120 грамматик, описанных в платформе ANTLR (Another Tool For Language Recognition). Проведенные тестирования показывают, что метод генерации случайных тестов учитывая обратную связь с фаззером, позволяет достичь большего покрытия кода, чем существующие методы.