通过模糊测试实现百万行代码的漏洞挖掘

张川[1]，毛慧[2]，王忠[3]，靳冬[3]，黄河清[1]，杨尚沅[2]
（1. 香港科技大学网络安全实验室，中国香港 999077；
2. 珠海南方软件网络评测中心，广东珠海 519085；
3. 中国合格评定国家认可中心，北京 100062）

摘 要： 在百万行代码的漏洞检测中，大规模程序的模糊测试备受挑战。在有限时间内生成满足输入语法和满足指数路径条件的指令是很困难的，而且触发漏洞的指令比挖掘代码的指令要复杂得多。研究一种高效指令生成方法，采用了强化学习进行语法推理，进行了优化的符号分析，在提高指令生成有效性的基础上，显著减少随机指令的执行时间。同时，对基于成分分析的模糊测试工作流程进行优化，以减小程序的规模，在保证效率的同时，为后期验证省去了不必要的程序片段。成功开发了新的指令生成方法，实现了发现大规模程序中的漏洞的目标。在现有基准上开展了测试，证明了方法的有效性。

关键词： 模糊测试；指令生成；语法推理；符号分析

中图分类号： TP3-0 **文献标识码：** A **文章编号：** 2095-8412 (2019) 04-041-08

工业技术创新 URL: http: //www.china-iti.com DOI: 10.14103/j.issn.2095-8412.2019.04.008

INTRODUCTION

Fuzzing is one of the most powerful vulnerability detection techniques [1-5], and has been widely adopted to detect various kinds of software security issues in the industry. For example, since 2016, there are over 9,000 vulnerabilities found by OSS-Fuzz in widely-used third-party library [6]. One-third of Windows 7 Wex security bugs were found by Microsoft fuzzing service and millions of dollars were saved [7]. Heartbleed, which threatened the worldwide cyberspace in 2014, can now be detected within 10 seconds of fuzzing [8]. Since fuzzing can generate concrete inputs to trigger the vulnerability, it is very easy to form proof of concept (PoC) for developers to understand and fix the problems. The simple intuition of using swift random mutation to generate an enormous number of inputs for evaluating the target program makes fuzzing capable to deploy on all kind of projects.

However, fuzzing is ineffective to detect vulnerabilities in large-scale programs. There are three challenges brought by the large program size. First of all, it is difficult to generate well-format inputs containing diverse semantic. Some large programs such as MySQL require the inputs to follow its unique syntax standard. Malformed inputs cause the execution to stop at an early stage without running any core function of the program. Moreover, it is hard to produce inputs with diverse semantic for thoroughly evaluating all the functions. These different specifications from tremendous projects make it impossible for fuzzing to automatically produce effective inputs. Second, it is inefficient to generate inputs satisfying the large and complex path conditions. Along with the increasing program size, the complex path conditions narrow down the feasible input space. This reduces the possibility for random mutation to generate inputs satisfying the path conditions. Therefore, inefficient input generation results in many meaningless executions. This problem aggravates when the

program size gets larger. The third challenge comes from the execution overhead. Fuzzing uses instrumentation to record the execution information to guide the later input generation. According to recent studies [9-10], we notice that the overhead in existing instrumentation methods can be higher than 600%. Meanwhile, state-of-the-art fuzzing frameworks run target program millions of times to increase the probability of detecting vulnerabilities. This accumulated overhead makes fuzzing inefficient to deploy on the large-scale program. To solve these challenges for fuzzing, we propose our techniques to improve the efficiency for fuzzing so that it can better support the large programs.

1 RELATED WORK

1.1 Reinforcement Learning for Semantic Inference

It is difficult to generate inputs containing semantics for vulnerability. The key reason is the lack of effective guidance for input generation towards semantics. Majority of the existing works still use program coverage to judge the value of inputs. There is no clear standard to prioritize the inputs based on the semantics. Although they try to learn the semantic either from numerous of inputs [11] or manually given specifications [12], for large-scale programs, describing the whole possible semantics precisely is challenging with a limited number of feasible inputs. Meanwhile, it is also difficult to have a general model which suits for different specifications.

We propose our method to solve this problem based on the inspiration of reinforcement learning. Reinforcement learning [13] is a type of machine learning technique that enables the agent to learn in an interactive environment by trial-and-error using feedback from its actions and experiences. Unlike other machine learning techniques that seek the classification or similarity results, reinforcement learning algorithms aim at finding the optimal solution for maximizing the predefined reward. Depending on the target goals, the reward can be various. If we consider the vulnerabilities as certain types of program semantics satisfying the specific property, then we can design the reward standard of the learning algorithms to eventually generate the proper inputs satisfying such property. For example, if we want to detect memory leak vulnerabilities, then it is more wisely to use the operations that can cause more memory usage. Since fuzzing needs a fitness function to guide the input generation, this reward standard should be the new effective guidance to improve the quality of inputs. Compared with existing works, reinforcement learning combining with deep learning is more capable to handle large scale of inputs or different specification.

1.2 Symbolic Analysis

Symbolic execution is one of the most powerful methods to detect vulnerabilities [14-16]. It can precisely generate inputs that lead the execution to certain program points. For example, single condition $x \times x=100$ might be very difficult for random mutation to generate the feasible value $x=10$. The possibility is less than 10~20 for the integer variable x. The large-scale program aggravates this problem since the number of path conditions increases. Constraint solving makes up the weakness of random mutation in handling complex path conditions. The combination of the two methods [17, 4] enhances the ability of vulnerability detecting.

However, the effectiveness is mainly restricted by the well-known performance issue in constraint solving. The state-of-the-art fuzzing frameworks still need random mutation to generate the majority of the inputs based on the solver results. Yun et al. [5] use fewer constraints to generate inputs while leveraging the fuzzing to validate the input effectiveness. Dutra et al. [18] propose to generate multiple inputs to increase the possibility of satisfying more path conditions. These methods focus on the efficiency of generating more concrete inputs for fuzzing from solver directly. Our proposed method takes the benefit from these methods while leveraging the

constraint to provide extra guidance for fuzzing. Constraints themselves contain the information of related input variables. Simple generating inputs cannot well represent the whole information in the constraints. Our methods analyze the constraints and improve the effectiveness of mutation and thus make it more efficient for large-scale programs.

1.3 Compositional Fuzzing

Directly analyzing the entire program is not practical when the program size is very large. Furthermore, vulnerable snippets only represent a small portion of the program. It is not necessary to thoroughly analyze the program as a whole. Therefore, fuzzing should adopt compositional analysis to become more scalable. State-of-the-art compositional analyses mainly focus on symbolic execution [19] and unit testing [14]. We leverage the existing methods to discover local vulnerabilities. We combine the semantic inference method to generate input satisfying local syntax while using constraint solving to provide the more feasible inputs. The compositional fuzzing sets up the potential goals for whole-program fuzzing to increase efficiency.

Symbolic analysis can assist whole-program fuzzing to validate the goals set by compositional fuzzing. There has been significant works on precondition inference in the programming language community [20-25]. The main differences lie in that they aim for sound verification, while we target on speeding up fuzz testing. They infer a sufficient precondition for program safety and prove the correctness by showing the condition held for all program paths. In our method, we do not aim to infer the symbolic condition for all possible program executions. Instead, we infer multiple necessary conditions for the target program points. If related variables violate the necessary conditions at the early stage, there is no need to continue the execution so that they cannot go towards target program points.

2 PROPOSED TECHNIQUES

In this section, we describe the high-level views of our model and the workflow of our proposed methods.

2.1 Compositional Fuzzing

The given Figures 1 and 2 show the basic model of our proposed compositional framework. We leverage compositional fuzzing to provide goals for whole-program verification.

The other two engines, Semantic Inference and Symbolic Analysis, render the input with designed semantics and values. Now we explain some details behind each engine.

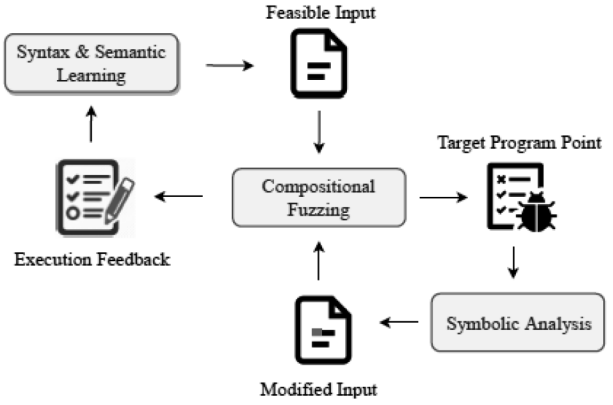

Figure 1 First Phase: Compositional Fuzzing

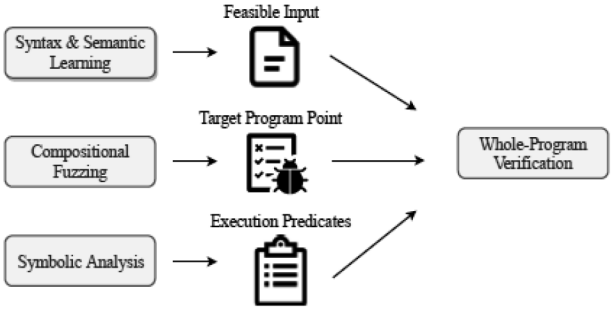

Figure 2 Second Phase: Whole-Program Verification

2.2 Semantic Inference Engine

Figure 3 shows the workflow of the semantic inference engine. We adopt the reinforcement learning to produce inputs containing the semantics needed. To make it more practical, we start by learning the different influences brought by the symbols and operations defined in the grammar. Understanding these semantics is more reasonable to generate inputs satisfying the properties provided by the vulnerability description. There is another benefit of combining the fuzzing with reinforcement learning. The

numerous time of executions provides abundant data with precise oracle for learning to understand the semantics. The learning model keeps updating during fuzzing which provides the more effective inputs that may trigger the vulnerability. There are many data that can be collected as oracle without many efforts such as execution time and memory status. State-of-the-art fuzzing frameworks [26-27] use these statuses to guide the input generation. However, how to properly mix different standards influences the input generation. Our reinforcement learning method is driven by target vulnerabilities and thus capable to find the inputs that maximize the results for all standards. Therefore, this should be the correct direction to solve the semantics problem.

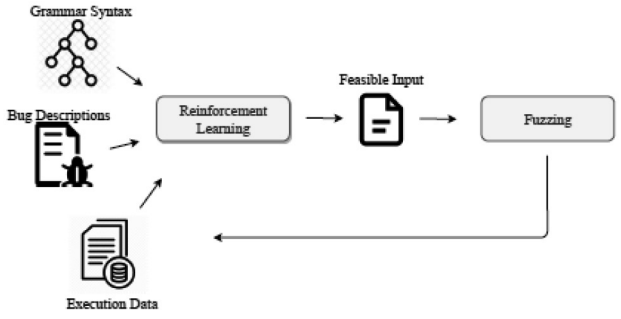

Figure 3 Semantic Inference

2.3 Symbolic Analysis Engine

As mentioned earlier in section 2, the symbolic analysis engine has two major functions. First, the symbolic analysis provides extra guidance for fuzzing input generation (Figure 4). We notice that constraint contains lots of useful information such as related variables and feasible value ranges. Only generating feasible inputs cannot deliver this information to fuzzing. To provide more precise and useful guidance from constraint solving, we leverage the solver to analyze the constraint to extract extra information and transmit them to fuzzing. With the precise guidance, both composition and whole-program fuzzing can become more efficient in generating inputs.

Second, symbolic analysis dynamically instruments the assertions to halt unnecessary

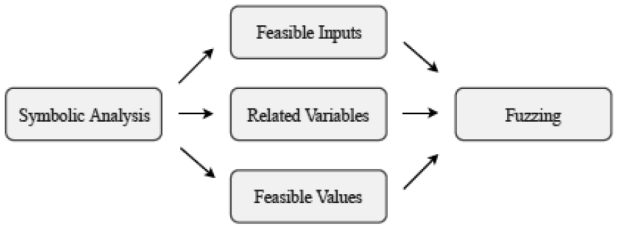

Figure 4 Symbolic Analysis for Fuzzing Guidance

executions (Figure 5). The symbolic analysis takes the concept of compositional evaluation and splits paths towards the target program points into multiple small units. We generate the necessary conditions for each unit and insert them into the programs. If related variables violate these conditions, the execution stops directly and starts with other inputs. Therefore, fuzzing can evaluate more possible inputs and find potential vulnerabilities.

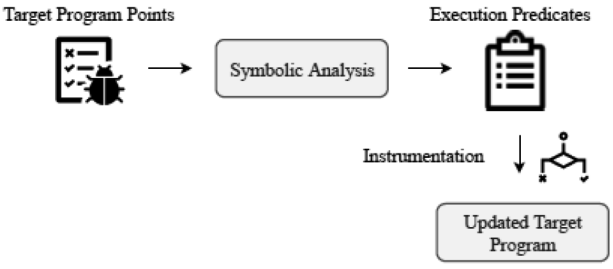

Figure 5 Symbolic Analysis for Execution Guidance

3 EVALUATION PLAN

All the evaluations share two metrics: code coverage and several vulnerabilities found within the same amount of time. There is some other supplementary evaluation standards for each method.

Comparison with other grammar-based fuzzers. To evaluate the effectiveness of semantics inference, we conduct the experiment with other grammar-based fuzzing [11-12] on projects such as MySQL, FFmpeg, Libav. All these projects require the input satisfying certain syntax. We also evaluate the effectiveness brought by different reward standards. Moreover, we need to optimize the parameters to make the model practical for all kind of projects.

Comparison with constraint-based fuzzers. In this experiment, we examine the code covering

and bug hunting capabilities of FuzzFuzzFuzz on the standard Lava-M dataset and real-world projects such as Binutils, OpenSSL. We compare our method with the state-of-the-art fuzzers that do not analyze the program semantics such as AFL [1], AFLFast [28], QSYM [5], Driller [4].

We demonstrate the effectiveness of our approach on the Lava-M [29] dataset, shown in Table 1. The number on the left shows the number of bugs found by fuzzing. Another one is the total number of bugs.

Table 1 Experiment results on Lava-M

	uniq	base64	md5sum	who
Our work	30/28	48/44	59/57	1600+/2136
AFL	9/28	0/44	0/44	1/2136
AFL-laf	24/28	28/44	0/57	2/2136
Steelix	7/28	43/44	28/57	194/2136
Vuzzer	27/28	17/44	Fail	50/2136
T-fuzz	26/28	43/44	49/57	63/2136
QSYM	28/28	44/44	57/57	1238/2136
Fuzzer	7/28	2/44	7/57	0/2136

This evaluation result shows that our optimized method outperforms other fuzzing frameworks. It even found some previous unseen bugs in the benchmark. For the program “who”, FuzzFuzzFuzz significantly outperforms all other fuzzers. Since “who” is much larger than the other 3 projects, it also proves the effectiveness of our method.

The effectiveness of compositional fuzzing. To evaluate the new framework analysis, we directly compare our method with other frameworks. We conduct the experiment with the same amount of time in different real-world projects. All the vulnerability counted as true positive must be confirmed by the developers.

Case study. We pick the vulnerability cases found by our new fuzzers to demonstrate the effectiveness of our works. We also analyze the program if our methods fail to outperform other frameworks to discuss the weaknesses of our methods.

4 PROGRESS AND PROSPECTS

Currently, we have implemented a prototype combining with symbolic analysis. In Section 3, we demonstrate the evaluation result compared with state-of-the-art fuzzing on the Lava-M benchmark. The results prove the effectiveness of the methods proposed in the paper. We are going to improve the prototype and evaluate the result on real-world projects.

Afterward, we will adopt the symbolic analysis result with dynamic instrumentation techniques such as Dyninst [30]. Our prototype first applies compositional analysis to gather information for potentially vulnerable points. These program points are set as the goals for later validation. The symbolic analysis optimizes the way of instrumentation and generates potential input towards these points. This work aims at effective guiding fuzzing execute towards the target program points and reducing the overhead caused by unrelated execution. These benefits make fuzzing more efficient for large-scale programs.

At last, we will adopt the reinforcement learning algorithm for different grammars such as SQL, HTTP to build the grammar-based fuzzing framework. It will also combine with compositional symbolic analysis to adjust the minor part of the input for specific value requirements. With all methods implemented, our final framework should be capable of detecting all kinds of vulnerability in the million lines of code projects.

FUNDING

This paper is funded by the National Key Research and Development Project (2016YFF0204002)

REFERENCES

[1] 2014. AFL: American fuzzy lop. http://lcamtuf.coredump.cx/afl/. (2014). Accessed: 2014.

[2] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by

Principled Search. In 2018 IEEE Symposium on Security and Privacy (SP), Vol. 00. 711–725. https://doi.org/10.1109/SP.2018.00046

[3] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano. 2014. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. 323–332. https://doi.org/10.1109/ICST.2014.45

[4] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In NDSS, Vol. 16. 1–16.

[5] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[6] 2016. OSS-FUZZ report. https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html. (2016). Accessed: 2016.

[7] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[8] 2015. libfuzzer. https://llvm.org/docs/LibFuzzer.html. (2015). Accessed: 2015.

[9] Stefan Nagy and Matthew Hicks. 2019. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In IEEE Symposium on Security and Privacy (Oakland).

[10] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In Proceedings of the 2017 ACMSIGSAC Conference on Computer and Communications Security (CCS ’17). ACM, New York, NY, USA, 2313–2328. https://doi.org/10.1145/3133956.3134046

[11] J. Wang, B. Chen, L. Wei, and Y. Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In 2017 IEEE Symposium on Security and Privacy (SP). 579–594. https://doi.org/10.1109/SP.2017.23

[12] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2139–2154.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. Nature 518, 7540 (Feb. 2015), 529–533. http://dx.doi.org/10.1038/nature14236

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. SIGPLAN Not. 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[15] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/

[16] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed symbolic execution. In International Static Analysis Symposium. Springer, 95–111.

[17] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE ’07). IEEE Computer Society, Washington, DC, USA, 416–426. https://doi.org/10.1109/ICSE.2007.41

[18] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In Proceedings of the 40th International Conference on Software Engineering (ICSE ’18). ACM, New York, NY, USA, 549–559. https://doi.org/10.1145/3180155.3180248

[19] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In Proceedings of the 40th International Conference on Software Engineering (ICSE ’18). ACM, New York, NY, USA, 350–360. https://doi.org/10.1145/3180155.3180251

[20] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In ACM SIGPLAN Notices, Vol. 51. ACM, 789–801.

[21] Satish Chandra, Stephen J Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions.

ACM Sigplan Notices 44, 6 (2009), 363–374.

[22] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic inference of necessary preconditions. In International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, 128–148.

[23] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition inference from intermittent assertions and application to contracts on collections. In International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, 150–168.

[24] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In ACM SIGPLAN Notices, Vol. 47. ACM, 133–146.

[25] Francesco Logozzo, Shuvendu K Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification modulo versions: Towards usable verification. In ACM SIGPLAN Notices, Vol. 49. ACM, 294–304.

[26] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018). ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[27] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS ’17). ACM, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[28] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coveragebased Greybox Fuzzing As Markov Chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS ’16). ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[29] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti,W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In 2016 IEEE Symposium on Security and Privacy (SP). 110–121. https://doi.org/10.1109/SP.2016.15

[30] 2014. Dyninst. https://dyninst.org/dyninst. (2014). Accessed: 2014.

作者简介：

张川（1974—），男，中国香港人，香港科技大学副教授。研究方向：软件安全、代码安全。
E-mail: charlesz@cse.ust.hk

毛慧（1988—），通信作者，女，湖北天门人，高级工程师。研究方向：代码安全。
E-mail: 576467379@qq.com

王忠（1974—），男，北京人，高级工程师。研究方向：软件和信息安全能力验证。
E-mail: wangzh@cnas.org.cn

靳冬（1982—），男，山西人，研究员。研究方向：软件、信息安全检测和认可。
E-mail: jind@cnas.org.cn

黄河清（1993—），男，中国香港人，香港科技大学在读博士生。研究方向：软件安全、信息安全。
E-mail: hhuangaz@cse.ust.hk

杨尚沅（1994—），男，广东珠海人，工程师。研究方向：代码安全。
E-mail: 1533231845@qq.com

（收稿日期：2019-07-05）

Generating PoCs for Million Lines of Code Programs via Fuzzing

ZHANG Charles[1], MAO Hui[2], WANG Zhong[3], JIN Dong[3], HUANG He-qing[1], YANG Shang-yuan[2]
(*1. Cybersecurity Lab, Department of Computer Science & Engineering, HKUST, Hong Kong 999077, China;*
2. Zhuhai Southern Software and Network Evaluation and Testing Center, Zhuhai, Guangdong 519085, China;
3. China National Accreditation Service for Conformity Assessment, Beijing 100062, China)

Abstract: Fuzzing large-scale programs is always challenging in nowadays vulnerability detection. It is difficult to generate inputs satisfying both input grammar and exponential path conditions within a limit amount of time to cover the million lines of code programs. Moreover, the inputs for triggering the vulnerability are much complicated than those achieving program coverage. Therefore, an efficient input generation method satisfying all the requirements becomes an urgent need. To address this issue, we adopt reinforcement learning for grammar inference and optimized symbolic analysis for enhancing the effectiveness of input generation. This effectiveness can significantly reduce the times of execution with random inputs. In the meantime, we optimize the workflow of state-of-the-art fuzzing inspired by compositional analysis to reduce the analyzed program size. Small program size guarantees efficiency while pruning out the unnecessary program pieces for later validation. The goal of this paper is developing a novel input generation method that can effectively generate input triggering the vulnerability in large-scale programs. We have done some experiments on the existing benchmark that show the effectiveness of our methods.

Key words: Fuzzing; Input Generation; Grammar Inference; Symbolic Analysis