A Survey of Hybrid Fuzzing based on Symbolic Execution

**Abstract**: Fuzzing has now developed into an efficient method of vulnerability mining. Symbolic execution is also a popular software vulnerability mining technology. Both are research hotspots in the field of network and information security. Hybrid fuzzing is the addition of symbolic execution technology on the basis of traditional fuzzing, and has now developed into a new branch of fuzzing. This article studies the existing hybrid fuzzing methods, reviews the development and evolution process and technical core of hybrid fuzzing, and compares the performance of currently well-known hybrid fuzzing through an experimental method based on symbolic execution. Finally, it discusses the existing problems in the field of hybrid fuzzing testing, and tries to look forward to its future development trend.

**Key Words**：Hybrid fuzzing, symbolic execution, Fuzzing

1. **Introduction**

Fuzzing is currently the most popular automated software vulnerability mining tool. By inputting a large number of random test cases into the program to be tested (test program after instrumentation processing), it can obtain test information such as coverage rate and possible crashes. information. Fuzzy testing has the advantages of high degree of automation, low consumption rate, and high utilization rate of computing resources. It is a dynamic analysis technology of programs. It has been proved by many practices to have good effects [5], [6], [7], and has been obtained in academia and industry. widely used. In terms of vulnerability mining, fuzzing has played a very amazing effect. Take AFL (America Fuzzy Lop, hereinafter referred to as AFL) [5], a typical mutation-based fuzzing tool as an example. Since its development in 2014, it has been discovered Hundreds of CVEs, this does not include the CVE vulnerabilities found in many AFL-based papers[8][9][10], and undisclosed vulnerabilities. However, the high degree of automation also leads to the fact that fuzzing cannot test all possible paths of the program to be tested, and cannot cover all the states of the program to be tested. Therefore, in the face of complex branching applications, it can only reach the depth of the path as soon as possible. To the breadth of full coverage.

Symbolic execution is a program analysis technology. The key idea is to use symbols to represent the input of the program instead of specific values, and then analyze the program to get the program input, and ensure that the input can be executed by a specific area code [16] . If the program to be tested is visually displayed in the form of a symbolic execution tree, it can be easily understood. Symbolic execution has great advantages in front of complex branches. Through program path exploration, it can break through complex branches in shallow locations and achieve programs. The state is fully covered, but the deeper path cannot be reached due to problems such as computational constraints. Obviously, fuzzing and symbolic execution have their own advantages and disadvantages. Hybrid fuzzing technology combines the advantages of both to reach a deeper path and obtain higher coverage. Hybrid fuzzing is better than using symbolic execution or fuzzing alone [3], [4].

This article will review the development of hybrid fuzzing, an automated software vulnerability mining technology, discuss key technologies, summarize typical tools in the process of technological development, and specifically use the CGC data set and the LAVA data set as the standard to compare nearly five The method proposed by the annual summit will organize experimental data and analyze horizontally. Finally, it discusses the future development direction of hybrid fuzzing, and summarizes the article.

1. **Technical overview**

**2.1. Vulnerability mining technology**

There are many technologies currently used for vulnerability mining, and there are three main ones: static analysis, dynamic analysis and symbolic execution [12]. These three technologies have their own advantages and disadvantages.

Static analysis is a software testing technology that analyzes the program to be tested without executing it. Commonly used analysis technologies include lexical analysis, syntax analysis, abstract syntax tree analysis, semantic analysis, control flow analysis, data flow analysis, Stain analysis, invalid code analysis, etc., are widely used in vulnerability mining. The advantage of static analysis is that it can quickly analyze tens of thousands of lines of code, and use formal methods to give provable and complete analysis results [19]. The shortcomings of static analysis are also obvious. The analysis results fail to give accurate POC (Sample of Vulnerability Triggering Proof). The purpose of analysis is to discover software vulnerabilities, but the obtained analysis results cannot generate input that can trigger vulnerabilities, such as PNG The image-reading program to be tested uses static analysis to obtain the result, but cannot obtain the PNG image that can trigger a certain vulnerability. You need to construct the PNG file yourself based on the analysis result.

The dynamic analysis can execute the program to be tested, and usually uses the instrumentation method to obtain various paths, and the coverage information can be obtained through the path. Different from static analysis, dynamic analysis can monitor the running of the program to be tested. When the program has a running error, the input that triggered the error will be saved. If it is also used in the PNG image reading type to be tested program, it can be saved and triggered. The wrong corresponding PNG image file. Dynamic analysis can also detect dependencies that cannot be detected by static analysis, such as polymorphic dynamic dependencies. The shortcomings of dynamic testing mainly come from instrumentation technology. Instrumentation technology is to insert specific operations in the program to be tested to detect the program to achieve the purpose of testing. Therefore, the inserted part of the code (high-level language or programming language) must be A certain negative performance impact on the original program to be tested. Another disadvantage is that it is impossible to obtain full coverage, because the running of the program to be tested is based on user interaction or automated testing, which does not guarantee full coverage of all possible locations of the program.

The emergence of symbolic execution technology can be traced back to 1976, [] et al. proposed this idea [22]. Symbolic execution uses program interpretation and constraint solving techniques to generate program input to explore the state space of the program to be tested, thereby triggering vulnerabilities. However, because symbolic execution must cover all program paths as much as possible, for example, two paths must be generated for a conditional judgment statement to ensure full coverage of the current link, so it usually triggers a large number of paths in the program to be tested, that is, "path explosion ". [12]

Although the mainstream of vulnerability mining is the above three categories, current research no longer uses a single technology, but tends to combine multiple technologies. The initial design idea of ​​fuzz testing is dynamic analysis [11], which uses instrumentation technology to dynamically monitor the test program, but the current mainstream fuzz testing tools also add static analysis technology, such as T based on fuzz testing and stain analysis technology. -Fuzz[20].

**2.2. Fuzzing**

Fuzzing testing technology was first proposed by [M] et al. in 1995 to conduct security testing for UNIX systems and find many vulnerabilities in components [11]. The core of fuzzing is unintended input. Whether it is based on mutation or generated fuzzing, it is used to obtain abnormal input that is different from manual testing. Once the input triggers the program exception, it will be saved and finally exploited by the vulnerability. Personnel conduct analysis. The core process of fuzz testing is shown in Figure [], [Test Process]. For a complete fuzzing process, there are three things to be prepared: seed set, fuzzing tool, and program to be tested.

Seed set is the cornerstone of fuzzing. A large number of random inputs are derived from seed mutation technology. High-quality seeds are very important for the efficiency of fuzzing, so tools such as AFL will provide seed banks. For a fuzzing tool like [] that can be spontaneously seeded, there is no need to provide a seed set, but at the beginning of the test program of [], a predetermined seed will be generated to replace the role of the seed in the form of a data stream, so from Basically, seeds are essential.

Fuzzing testing tools are the core of the whole test. The current mainstream fuzzing testing tools must have two capabilities: one is the ability to generate a large number of random inputs. When the seed file (data stream) is obtained, a large number of inputs need to be generated quickly. The input will be transmitted to the program to be tested in the form of a data stream, and a certain path will be executed. If you want to traverse as many paths of the program as possible, you need to generate as many different inputs as possible; the second is to execute multiple The coordination ability of the program, although the fuzzing test is an endless loop, it does not mean that the second input test is performed after the end of a test, because a large number of inputs are generated on the basis of the seed file and the operations are performed in sequence , The execution stage of the program to be tested will temporarily store a large number of input data streams in memory, making the program execution a bottleneck in the process, and will seriously slow down the overall efficiency of fuzzing. Therefore, fast and effective execution of the program to be tested is the fuzzing technology The second important ability.

The program to be tested is the ultimate goal of fuzz testing. All tests are carried out around the program to be tested. There will be some formal differences in the program to be tested for different test targets, and manual intervention by researchers is required for different types of targets. Adjust accordingly. For example, for library testing, it is usually necessary for the tester to have a certain understanding of the source code, and then write different programs according to different target functions.

In the flow chart of the fuzzing test, there is room for improvement in each stage. For example, in the stage when the input is obtained from the seed set, mopt-afl [quote] is added; in. . . [Cite three examples].

**2.3. Hybrid Fuzzing Technology**

As mentioned above, the current fuzzing test has certain room for improvement in various stages. Many branches have been developed by combining other technologies. Among them, the hybrid fuzzing technology is a combination of fuzzing technology and symbolic execution technology. There is already a proof of work. Hybrid fuzzing technology has shown good results in automated vulnerability mining [Citation], and the combination of the two will achieve better results in code coverage and vulnerability discovery efficiency than using either alone.

1. **Research Status of Hybrid Fuzzing**

In 2007, [] et al. proposed the concept of hybrid testing, which combined symbolic execution and automated software testing. One of the advantages of symbolic execution is that all states can be found in the vicinity of the specified location. One of the advantages of fuzzing is to quickly dive into the state of the program. The author combines the two to propose a hybrid test method, and conduct experiments to prove The mixed effect is much better than the single case. In the test phase, by adding a mixed execution interface to the top level of the program design of CUTE [quote], a red-black tree structure program and VIM software were tested for vulnerability mining, and experimental data were given. The author did not clearly point out whether the two technologies compensate for each other's shortcomings, and from the experimental results, the hybrid test combines the advantages of the two, and the author did not improve the two technologies. Although the author did not explicitly propose hybrid fuzzing technology in this article, the random testing idea used in it is consistent with fuzzing, so this research became the conceptual prototype of many hybrid fuzzing tests later. In 2012, [] et al. first proposed the concept of hybrid fuzzing (Hybrid Fuzzing) [].

After the hybrid fuzzing test was proposed, it has not made great progress in the field of vulnerability mining and software testing. With the continuous improvement of the computing power of computers and the research of algorithms, symbolic execution technology has made certain breakthroughs in constraint solving and so on. Compared with symbolic execution, fuzzing has not made any landmark progress, and hybrid fuzzing has not made breakthrough progress. In 2015, the emergence of AFL [5] brought new ideas to the industry and academia. It was a milestone development of fuzzing and a turning point for hybrid fuzzing technology. The coverage-oriented idea brought design to hybrid fuzzing. Ideas, and the fork-server technology in AFL also provides implementation ideas for later hybrid fuzzing technology.

In 2016, N, J[] et al. proposed a new hybrid fuzzing technology-Driller. Different from the mixed test in 2007, Driller used the fork-server idea to combine the most efficient fuzzing technology [] and symbolic execution technology at the time, and realized efficient vulnerability mining for binary files. [Introduction to the core of the paper] Driller laid the development direction of hybrid fuzzing technology, fuzzing technology plus symbolic execution technology.

Hybrid fuzzing technology has exploded, and domestic and foreign researchers have been attracted. QSYM proposed by [] et al. in 2018, and SAVIOR in 2019 [] et al. In the same year, domestic research on hybrid fuzzing began to emerge. [] et al. of Wuhan University proposed a prototype system called DigFuzz, which deeply analyzed the advantages and disadvantages of hybrid fuzzing. Afleer [4], proposed by Xie Xiaofei, Li Xiaohong, etc., combines AFL [5] and KLEE [17] through a method based on branch coverage, using AFL to generate a large number of test cases, and then based on KLEE on the coverage information obtained by AFL. Search, use the search results to guide the generation of test cases, and get test cases that only cover uncovered branches. The author further confirms through the LAVA-M data set that the hybrid fuzzing technique will get better test results than using fuzzing or symbolic execution alone.

In 2020, hybrid fuzzing technology has become one of the most important coverage-oriented fuzzing branches. Researchers are beginning to be dissatisfied with the improvement of the combination method, and better serve the fuzzing technology by improving symbolic execution. PANGOLIN [quoted by [] from Xiamen University, H[] from Hong Kong University of Science and Technology and others [quoted] improved the efficiency of vulnerability mining by modifying the constraint solving part of symbolic execution. First, PANGOLIN proposed a polyhedral path solving method to improve the efficiency of constraint solving, and the result of this constraint solving was used to guide the mutation process of the fuzzing stage, and the compliance execution and the fuzzing process were more closely combined. Improve the efficiency of vulnerability mining.

1. **Further Directions**

In terms of fuzzing, the mode of fusion of multiple technologies has become the mainstream, as one of the most important branches: hybrid fuzzing technology, which is based on the combination of compliance execution technology and fuzzing technology, but essentially the main body of thought lies in fuzzing. test. Coverage-oriented fuzzing testing is also the most important one. It is often used to incorporate hybrid fuzzing testing technology. Therefore, it is possible to improve the combination method that is conducive to improving the coverage rate. For example, the use of instrumentation information for constraint solving can speed up the improvement of coverage in the fuzzing process. Therefore, it is the future trend to use compliance execution to increase the coverage rate, that is, maintaining the advantages of compliance execution in hybrid fuzzing and increasing the coverage rate of the original fuzzing test. In addition, how to better combine compliance execution and fuzz testing is also a development direction. A more efficient combination will increase the efficiency of the fuzzing program itself, thereby improving the efficiency of vulnerability mining.

1. **Conclusions**

Since the hybrid fuzzing test was proposed, it has experienced two upsurges. The first was officially proposed and named hybrid fuzzing. Since then, researchers have begun to pay attention to such a branch. The second time was to combine afl's driller to determine a mode that conforms to execution and fuzz testing. A large number of researchers began to pay attention to this type of research. Hybrid fuzzing technology has become one of the most important branches of fuzzing technology. With the development of compliance execution and the development of fuzzing, hybrid fuzzing technology will also have lasting development.

**Reference**

[1]. https://www.cnvd.org.cn/flaw/statistic

[2]. Shoshitaishvili Y , Wang R , Hauser C , et al. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware[C]// Network & Distributed System Security Symposium. 2015.

[3]. 【第一次提出混合测试】Majumdar, Rupak , and K. Sen . "Hybrid Concolic Testing." International Conference on Software Engineering IEEE, 2007.

[4]. 【国内混合模糊测试中文】谢肖飞, XIE XiaoFei, 李晓红,等. 基于符号执行与模糊测试的混合测试方法[J]. 软件学报, 2019, Vol.30Issue(10):3071-3089.

[5]. afl

[6]. honggfuzz

[7]. ossfuzz

[8]. collafl

[9]. mopt-afl

[10]. aflgo?aflfast?

[11]. 【第一次提出模糊测试】Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services

[12]. driller

[13]. QSYM

[14]. 【SAVIOR】Chen Y, Li P, Xu J, et al. SAVIOR: towards bug-driven hybrid testing[C]//2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020: 1580-1596.

[15]. Pangolin

[16]. Baldoni R, Coppa E, D’elia D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys (CSUR), 2018, 51(3): 1-39. 【符号执行综述】

[17]. KLEE 【】 Cadar C, Dunbar D, Engler D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//OSDI. 2008, 8: 209-224.

[18]. 【LLVM】Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004: 75-86.

[19]. 【静态程序分析】Chess B, McGraw G. Static analysis for security[J]. IEEE security & privacy, 2004, 2(6): 76-79.

[20]. 【T-Fuzz污点分析模糊测试】Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: fuzzing by program transformation[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 697-710.

[21]. 【2012年混合模糊测试】Pak B S. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution[J]. School of Computer Science Carnegie Mellon University, 2012.

[22]. 【第一次提出符号执行】King J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.

[23]. 【2019武汉大学的混合模糊测试】