Survey of Directed Fuzzy Technology

Yan Zhang and Junwen Zhang
School of Computer and Information Technology
Beijing Jiaotong University
zjw@bjtu.edu.cn

Dalin Zhang
National Research Center of Railway Safety Assessment
Beijing Jiaotong University
dalin@bjtu.edu.cn

YongminMu
School of Computer
Beijing Information and Technology University
yongminmu@163.com

Abstract—**The** fuzzy testing technology can effectively detect vulnerabilities. Based on Directed Symbolic Execution (DSE) fuzzing and Directed Grey Box Fuzzing (DGF), which can reach the specified target locations and scan the vulnerability quickly and efficiently. This paper introduces the theoretical knowledge of directed fuzzy testing technology, and several state-of-the-art fuzzy testing tools to elaborate the prlnclple, advantages, disadvantages and the prospect of directed fuzzy technology.

*Keywords- **fuzzy** testing; directed symbolic execution; directed grey box fuzzing; **fuzzy** tools*

I. INTRODUCTION

Software security information is critical to understanding the security issues and distribution of software. According to a report published by the well-known security company Veracode in 2017 [1], today, agile development and open source software are popular. An average of 75% of software code in a software comes from open source components, but vulnerabilities in these open source components bring a huge security risk. At the same time, most of the information about security comes from the protection technologies of firewalls or anti-virus networks, gateways or terminals provided by "perimeter defense companies". These techniques only focus on known vulnerabilities and involve fewer a-day vulnerabilities hidden in binary software.

Fuzzy testing is a security test technology, which uses many semi-effective data as input to the application, and uses the program as an indicator to find out the possible security vulnerabilities [2] in the application. The fuzzy testing is divided into the following three types [3]. Black box Fuzzing, which only needs to analyze the results of the program execution [4, 5, 6]; White Box Fuzzing [7], which requires heavyweight level program analysis and constraint solving; Grey Box Fuzzing (GF) is somewhere in between, using lightweight tools to collect certain program structures and constrain solving at compile time. However, existing grey box fuzzers cannot be directed effectively.

Directed fuzzing is a vulnerability detection technology that focuses on the location of a target in a user-specified program. It is mainly divided into DSE and DGF. When the program changes, we are more concerned about whether this change introduces other vulnerabilities. The directed fuzzer uses this change as the target location for vulnerability scanning to detect more vulnerabilities and prevent them from spreading widely. Unlike non-directed fuzzers, directed fuzzers spend a lot of time on finding a specific target location instead of wasting a lot of time on unrelated program modules. Directed fuzzing, as an improvement to traditional random testing, uses stain analysis to locate seed input areas that can affect security-sensitive program points, and it focuses on changing these seeds to reveal defects and vulnerabilities.

Most existing directed fuzzers are based on symbolic execution [8]. The goal of a stain-based directed fuzzer is to identify and fuzz specific input bytes in the seed to obtain a particular value at a given program location. It uses classical stain analysis [9] to identify certain parts of the seed input that should be prioritized to increase the probability of generating the value required to observe the target location vulnerability (for example, the zero value in the denominator of the division operator [10]), this can greatly reduce the search space. It does not require heavyweight symbolic execution and constraint solving mechanisms. However, the user is required to provide a seed input that has reached the target location. DGF is a vulnerability detection technology that implements orientation based on GF, and the user can specify multiple target positions at a time. At the same time, the user can give an initial seed input or an empty input, and the directed grey box fuzzer can be fuzzed.

At present, the mainstream fuzzers are as follows, as shown in Table 1.

TABLE!. SUMMARIES OF THE TYPICAL FUZZERS

Name	*Birth Year*	*KeyTechniques*
Peach	2004	Black box fuzzing
TaintScope	2010	Symbolic execution
AFL	2013	Grey box fuzzing
KAcH	2013	Symbolic execution
SDFuzz	2014	Directed fuzzing
SeededFuzz	2016	Directed fuzzing
Kelinci	2017	Grey box fuzzing
AFLGo	2017	Directed grey box fuzzing
ColIAFL	2018	Grey box fuzzing
SAFL	2018	Directed grey box fuzzing/ directed symbolic execution
PTfuzz	2018	Grey box fuzzing

978-1-5386-6565-7/18/$31.00 ©2018 IEEE

II. DIRECTED SytvlBOLIC EXECUTION

A. Symbolic Execution Algorithm

Symbolic execution was originally used in the field of compilers, program understanding, etc. [11]. The main idea is to symbolize the input of the program, and execute these symbol variables as input, collect the predicate constraints of the branch condition judgment, and finally use the solver to get the new test input [12].

The most traditional symbolic execution is static symbolic execution. First, the input variable is symbolized, that is, the input variable is set to x. Then through the static analysis program, it is converted into an intermediate language, and the symbolized variable is changed in the program flow, thereby outputting a value of the symbolized variable. An example is shown in Figure 1 [13]:

```
a = raw_inputO;
b = 2*a;
if (b = = 10)
    print "win";
else
    print "lose";
```

Figure 1. Simple code example

In this code example, the traditional run is to enter the value of a, then run the following code. In the process of static symbolic execution, a is first symbolized, e.g. a = x, b = 2*x. When b == 2 * x, it enters the 'win' path; ifb != 2 * x, it enters the 'lose' path. The states in which the two paths are combined is called the execution tree, b == 2 * x and b != 2 * x are the path constraints. When the symbolic execution ends (that is, the program is normal or abnormally exited), the constraint solver will solve the path constraint, and the solution is the value required to go to this path. On another level, symbolic execution is the transformation of reachability problems into constraint solving problems.

Of course, this approach looks perfect, but there are many problems in the actual implementation process, such as the constraints cannot be solved. Combining traditional static symbolic execution with actual execution is called dynamic symbolic execution [14]. Dynamic symbolic execution maintains two states: one is the state of the actual variable, and the other is the symbolized state. The actual state maps randomly generated values to variables, while the symbolized state symbolizes variables. The dynamic symbolic execution first runs according to the actual state, and collects the constraints of the symbolization of the variables actually running to the path and solves them. Then, the constraint is inverted, and the constraints of the other path are obtained and solved. The process repeats until all paths have been explored or the user-set limits are reached.

As shown as Figure 1, the dynamic symbolic performs a randomly generated variable a = 7, which enters the 'lose' path when actually executed. In the judgment statement, another path can be obtained by negating the collected constraints. Therefore, in this way, the problem that the constraint cannot be identified or solved is avoided. Zhe Chen et al. extract behavior information from the original complex Control Flow Graph (CFG) by using dynamic symbolic execution, and then add constraints to the control flow model to present a control flow-based Extended Program Behavior model with Finite State Machine control parameters (EPBFSM). Finally, a new fuzzy input is generated by solving the constraints generated by EPBFSM. This approach not only finds possible path-aware vulnerabilities, but also identifies possible access control object-aware vulnerabilities [15].

B. Directed Symbolic Execution

DSE translates the reachability problem into an iterative constraint solving problem. Since most paths are not feasible, the search is typically iterative by finding a feasible path to the intermediate target. For example, the patch test tool K_{ATCH} [16] uses the symbolic actuator KIIIE to reach the part of the program changes. K_{LEE} [17] uses the method of symbolic execution to systematically explore the state space of feasible paths by heavy-weight analysis of program analysis and constraint solving [18]. Once a feasible path that can actually reach the target location is identified, the test case is generated a posteriorly as a solution to the corresponding path constraint.

```
55 /* Read type and payload length first */
56 hbtype = *p++;
57 n2s(p, payload);
58 pl = p;

65 if (hbtype == TLSl_HB_REQUEST) {
77 /* Enter response type, length and copy payload */
78 *bp++ = TLS 1_HB_RESPONSE;
79 s2n(payload, bp);
80 memcpy(bp,pl,payload);
```

Figure 2. Heartbleed commit[19]

Suppose that line 80 is set as the target and K_{ATCH} finds a feasible path π_0 to reach line 65 as the intermediate target. Next, K_{ATCH} solves the constraint $\varphi(\pi_0) \cap$ hbtype == TLSl_HB_REQUEST condition by the constraint solver to generate an input that can be executed to the line 80 (target position). Unlike grey box fuzzers, white box fuzzers based on symbolic execution provide a simple handle to achieve directed fuzzing. Most patch testing tools are based on DSE.

However, the effectiveness of DSE sacrifices its efficiency [20]. DSE takes a long time to perform program analysis and constraint solving. During each iteration, DSE uses program analysis to identify those branches that need to be negated to get closer to the target, constructing the corresponding path conditions based on the sequence of instructions along those

paths, and use a constraint solver to verify the satisfiability of these conditions.

DSE has been used to reach dangerous locations in the program, overwrite changes in the patch, overwrite previously uncovered program elements to increase coverage, verify static analysis reports, abrupt detection, and reproduce field failures.

III. DIRECTED GREY BOX FUZZING

A. Grey Box Fuzzing

Coverage-based fuzzing is designed to generate input that can achieve maximum code coverage, as well as the ability to explore deeper program paths. Saahil Ognawala et al. proposed an open source framework Munch [21] that implements two hybrid techniques based on fuzzing and symbolic execution, which achieves functional coverage of deeper paths.

AFL [22] is a very good grey box fuzzer for C programs and has the ability to discover many high-impact vulnerabilities [23] and can be extended. It builds test cases based on code instrumentation, resulting in high availability of use cases, and many performance optimizations for Linux that make it very fast. AFL is easy to extend, and Kelinci [24] is a tool for linking AFL to instrumented Java programs. It does not require modification of AFL and is easy to parallelize. This tool implements the possibility of applying AFL-type fuzz testing to Java programs and can prove its effectiveness. AFL gets the input program from the file or standard input 'stdin' and makes a good fuzzing, but the program on the network is not convenient and needs some support from the auxiliary library. CollAFL [25] uses the application as input and uses three fuzzy strategies to provide more accurate coverage information to mitigate path conflicts and improve the speed of discovering new paths. Among them, three are untouched neighbor descendants preferentially fuzzed; untouched neighbor branches are preferentially fuzzed; seed priority fuzzy with more access memory operations. SAFL [26] is based on KLEE and AFL to generate seeds that can explore deeper paths earlier and easier, and proposes an algorithm that can perform a rare or deeper path with higher probability.

B. Directed Grey Box Fuzzing

DGF maps the reachability problem to an optimization problem and uses a specific meta heuristic algorithm to calculate the minimum distance from which the seed is generated to the target. AFLGo is a directed grey box fuzzer that is an extension of AFL.

AFLGo [27] can expose more vulnerabilities in less time [28]. While the DSE generates a single input, the grey box fuzzer can generate more orders of magnitude of input. Starting with the same seed input, the directed grey box fuzzer AFLGo takes less than 20 minutes to expose the Heartbleed vulnerability; however, the DSE tool KATeR does not expose the Heartbleed vulnerability for 24 hours.

DGF preserves the efficiency of GF because all program analysis is in the compile phase. At the same time, DGF can be easily parallelized, allowing more computing power to be allocated in order to detect vulnerabilities quickly and efficiently. AFLGo is easy to set up with just a few thousand lines of code and can be integrated with OSS-Fuzz to expose more security critical programs and libraries. BuzzFuzz [29] only gets the stain data of the target program, which is a good example of how the grey box fuzzer works. Compared to DSE, the DGF does not require heavyweight symbolic execution, program analysis, and constraint solving. The lightweight program analysis implemented by DGF is done at compile time.

The GF test uses the branch information collected at program run time as feedback to guide the selection of seeds. AFL and AFLGo detect and obtain branch information at compile time. QEMU emulation extended AFL can also obtain branch information of binary program. PTfuzz uses hardware mechanism (Intel processor trace) to collect branch information and obtained the performance equivalent to the compile-time detection method [30].

Seed input is very important for directed fuzzing. SeededFuzz [31] implements a seed selection method to cover more critical programs and detect more vulnerabilities. SDFuzz [32] searches for key function locations on binary software, uses stain analysis to classify input data into security-related data and security-independent data, and finally changes security-related data to implement directed testing. TaintScope [33] is a directed fuzzy tool that implements fine-grained dynamic stain tracking at the binary level, identifies key operations in the input, and identifies potential vulnerabilities. In addition, it is also a fuzzy test tool that supports checksums.

IV. CONCLUTION AND PROSPECT

Both DSE and DGF can reach the program-specified location. However, DSE has more heavyweight program analysis than DGF fuzzing. In the future, it is possible to integrate directed white box fuzzing and DGF based on symbolic execution, and to take advantage of the precise analysis of symbolic execution and the multi-input of grey box fuzzing generation, the integrated directed fuzzy technology will be able to take advantage of their comprehensive advantages to alleviate their individual weaknesses. At the same time, when choosing to execute test cases, you can use machine learning algorithms to determine the pros and cons of test cases and the order of priority of test case execution based on some models learned from past experience.

However, fuzzy testing also has certain limitations. For example, access control vulnerability, that is, the fuzzy testing does not recognize the permissions of the application itself. Bad design logic, that is, fuzzy testing does not identify whether the problem found is due to a security problem. The back door, that is, the fuzzy testing does not recognize whether it is a back door function. Destruction, that is, the SIGSEGV signal will cause the fuzzy testing to fail to identify whether the memory is corrupted. Multi-stage security vulnerabilities, that is, fuzzy testing is useful for identifying a single vulnerabilities, but they are not very useful for vulnerabilities that are caused by small chain of vulnerabilities. In the future, perhaps for these limitations, directed fuzzy techniques can give an answer.

At present, we customize the fuzzers according to input characteristics, mutation strategy, seed sample screening and abnormal sample discovery and analysis, which is expensive.

Yang Meifang integrated the grey box fuzzers such as AFL, Peach and Honggfuzz with the programmable fuzzy test framework into a programmable fuzzer, customized the service at a small cost, and achieved certain results [34]. The directed fuzzy technology still has a lot of application space. In the future, it will be a research hotspot to explore how to use it in the fields of defect detection, confirmation and test data generation [35,36,37] and so on.

ACKNOWLEDGMENT

The National Natural Science Foundation of China under Grant N0.61502029, the Beijing Natural Science Foundation (Z160002), Postgraduate Education (71D 1811013) and the Opening Project of Beijing Key Laboratory of Internet Culture and Digital (5221835409) sponsor this work. We are grateful to the anonymous reviewers for their comments on earlier drafts of this paper.

REFERENCES

[1] Software Security Report [M]. Veracode . 2017

[2] Honghui Li, Jia Qi, Feng Liu. Research on Fuzzy Testing Technology[J]. Science in China, 2014, 44(10): 1305-1322.

[3] Liang H, Pei X, Jia X, et al. Fuzzing: State of the Art[J]. IEEE Transactions on Reliability, PP(99):1-20.

[4] Website. 2017. Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/. (2017). Accessed: 2017-05-13.

[5] Website. 2017. SPIKE Fuzzer Platform. http://www.immunitysec.com. (2017). Accessed: 2017-05-13.

[6] Website. 2017. Zzuf: multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf. (2017).Accessed: 2017-05-13.

[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.

[8] Maria Christakis, Peter MOiler, and Valentin Wustholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). 144 155.

[9] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, Stephen McCamant, undefined, undefined, undefined, and undefined. 2017. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. IEEE Transactions on Software Engineering 43,2 (2017), 164 184.

[10] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In NDSS'17. 1 14.

[11] King JC. Symbolic execution and program testing[C]. Communications of the ACM. 1976:385-394.

[12] Weina Niu, Xuefeng Ding, Zhi Liu. Binary Code Vulnerability Discovery Based on Symbolic Execution[J]. Computer Science, 2013, 40(10): 119-121.

[13] Website. 2018. http://www.cnblogs.com/QxJDchen/p/9291335.html (2018).

[14] Jing An. Research on Key Techniques of Dynamic Symbol Execution [D]. Beijing University of Posts and Telecommunications, 2014.

[15] Chen Z, Guo S, Fu D. A Directed Fuzzing Based on the Dynamic Symbolic Execution and Extended Program Behavior Model[M]. 2012.

[16] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). 235–245.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08). 209-224.

[18] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In Proceedings of the 18th International Conference on Static Analysis (SAS'11). 95-111.

[19] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for Inhouse Debugging. In Proceedings of the 34th International Conference on Software Engineering (lCSE '12). 474--484.

[20] Marcel Bohme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. IEEE Transactions on Software Engineering 42, 4 (April 2016), 345-360.

[21] Ognawala S, Hutzelmann T, Psallida E, et al. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach[J]. 2017.

[22] Website. 2017. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. (2017). Accessed: 2017-05-13.

[23] Website. 2017. AFL Vulnerability Trophy Case. http://lcamtuf.coredump.cx/afl/#bugs. (2017). Accessed: 2017-05-13.

[24] Kersten R, Luckow K, Pasareanu C S. POSTER: AFL-based Fuzzing for Java with Kelinci[C]11 ACM Sigsac Conference. ACM, 2017:2511-2513.

[25] Gan S, Zhang C, Qin X, et al. CoIIAFL: Path Sensitive *Fuzzing[C]11* CoIIAFL: Path Sensitive Fuzzing. IEEE Computer Society, 2018:679-696.

[26] Mingzhe Wang,Jie Liang,Yuanliang Chen,Yu Jiang.SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing[C]. *IEEE*/ACM 40th International Conference on Software Engineering: Companion.2018.

[27] Marcel Bohrne, Van-Thuan Pham, Manh-Dung Nguyen, Abhik Roychoudhury.2017. Directed Greybox Fuzzing.In CCCS'17.

[28] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 1, Article 20 (Jan. 2012), 8 pages.

[29] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in Proc.USENIX Symp. Netw. Syst. Des. Implementation, 2016, pp. 275-289.

[30] Zhang G, Zhou X, Luo Y, et al. PTfuzz: Guided Fuzzing with Processor Trace Feedback[J]. IEEE Access, 2018, PP(99):1-1.

[31] Wang W, Sun H, Zeng Q. SeededFuzz: Selecting and Generating Seeds for Directed *Fuzzing[C]11* Intemational Symposium on Theoretical Aspects of Software Engineering. IEEE, 2016:49-56.

[32] Wu B, Zhang B, Wen S M, et al. Directed Fuzzing Based on Dynamic Taint Analysis for Binary Software[J]. Applied Mechanics & Materials, 2014, 571-572:539-545.

[33] Wang T, Wei T, Gu G, et al. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection[J]. 2010, 41(3):497-512.

[34] Meifang Yang, Wei Huo, Yanyan Zou. Programmable Fuzzy Testing Technology[J]. Joumal of Software.2018(5).

[35] Zhang D, Sui J, Gong Y. Large scale software test data generation based on collective constraint and weighted combination method[J]. Tehniki vjesnik, 2017, 24(4): 1041-1049.

[36] Zhang D, Jin D, Gong Y, et al. Research of alarm correlations based on static defect detection[J]. Tehnicki vjesnik, 2015, 22(2): 311-318.

[37] Zhang D, Jin D, Gong Y, et al. Diagnosis-oriented alarm correlations[C]IISoftware Engineering Conference (APSEC), 2013 20th Asia-Pacific. IEEE, 2013, 1: 172-179.