Hunting Deeply Hidden Software Vulnerabilities

Subjects: Computer Science, Information Systems

Contributor: Fayozbek Rustamov

Fuzz testing is a simple automated software testing approach that discovers software vulnerabilities at a high level of performance by using randomly generated seeds. However, it is restrained by coverage and thus, there are chances of finding bugs entrenched in the deep execution paths of the program. To eliminate these limitations in mutational fuzzers, patching-based fuzzers and hybrid fuzzers have been proposed as groundbreaking advancements which combine two software testing approaches. Despite those methods having demonstrated high performance across different benchmarks such as DARPA CGC programs, they still present deficiencies in their ability to analyze deeper code branches and in bypassing the roadblocks checks (magic bytes, checksums) in real-world programs. In this research, we design DeepDiver, a novel transformational hybrid fuzzing tool that explores deeply hidden software vulnerabilities. Our approach tackles limitations exhibited by existing hybrid fuzzing frameworks, by negating roadblock checks (RC) in the program. By negating the RCs, the hybrid fuzzer can explore new execution paths to trigger bugs that are hidden in the abysmal depths of the binary. We combine AFL++ and concolic execution engine and leveraged the trace analyzer approach to construct the tree for each input to detect RCs. To demonstrate the efficiency of DeepDiver, we tested it with the LAVA-M dataset and eight large real-world programs. Overall, DeepDiver outperformed existing software testing tools, including the patching-based fuzzer and state-of-the-art hybrid fuzzing techniques. On average, DeepDiver discovered vulnerabilities 32.2% and 41.6% faster than QSYM and AFLFast respectively, and it accomplished in-depth code coverage.

Keywords: software vulnerability ; hybrid fuzzing ; concolic execution ; patching-based fuzzing

1. Software Vulnerability

Software vulnerability is considered one of the foremost critical threats to any computer network and programs will inevitably have defects [1]. Many of these susceptibilities have the potential to exploit programs, often with malicious intent. Such imperfect codes pose critical threats to information security [2]. Therefore, it is essential to detect vulnerabilities that exist within components of a computer program.

Generally, it is accepted in the cybersecurity sphere that automated software analyzing approaches have achieved magnificent progress in discovering software vulnerabilities. In particular, fuzzing techniques have been applied [3][4] to discover popular program vulnerabilities [5][6], for instance, *“Month of Kernel Bugs”* [7], *“Month of Browser Bugs”* [8] and *“Heartbleed”* [9][10] bugs.

2. Fuzzing

Fuzzing is the most powerful automated testing technique that discovers security-critical vulnerabilities and security loopholes in any program cost-effectively and rapidly by providing invalid or random data that is generated and feeding them to the program [11][12][13]. It has advanced into a straightforward and efficient tool for the verification of code security and improvement of reliability. Moreover, fuzzing tools are widely used for various purposes. For instance, it is applied in *Quality Assurance (QA)* to analyze and secure internally developed programs, in *Vulnerability Assessment (VA)* to test and attempt to crack software package or system, and in *System Administration (SA)* to examine and secure a program in its usage environment.

Despite the abovementioned competencies of fuzzing, it still presents several weaknesses. The technique alone cannot deal with all security threats. To perform much more effectively in discovering serious security threats, it will require a significant amount of time [14]. It also has small-scale capabilities in achieving high code coverage.

To overcome the critical limitations of fuzzing, a hybrid approach called *hybrid fuzzing* [15] has been proposed recently, and after showing high-quality performance across various synthetic benchmarks, it has become increasingly popular in bug detection [16][17][18]. Research has shown that fuzzing and concolic execution [19] are powerful approaches in software

vulnerability discovery; combining them can possibly leverage their remarkable strengths and possibly mitigate weaknesses in the program.

The cardinal idea behind hybrid fuzzing is to apply the fuzzing technique in exploring paths and taking strategic advantage of concolic execution for the purpose of providing an excellent solution to path conditions. More specifically, fuzz testing is efficient in exploring paths that involve general branches, not to mention its ability to test programs quickly. However, it is not effective in exploring paths containing specific branches that include magic bytes, nested checksums. In comparison, concolic execution [19][20][21] extensively applied in detecting a significant number of bugs, can easily generate concrete test cases, but still has a path explosion problem. For example, Driller [18] provides a state-of-the-art hybrid fuzzing approach. It demonstrated how efficient the hybrid testing system was in DARPA's Cyber Grand Challenge binary tests, discovering six new vulnerabilities that could not be detected using either the fuzzing technique or concolic execution. In addition, as generally understood, the hybrid fuzzing approach is limited due to its slow concolic testing procedure. Hence, QSYM [22] is tailored for hybrid testing that implements a fast concolic execution to be able to detect vulnerabilities from real-world programs.

Patching-based fuzzers can address the issue of fuzzing approaches from a different angle. Although this method was introduced to science a decade ago, it has not only shown remarkable results but has also been commonly applied in software vulnerability detection. To give an example, T-Fuzz [23] has recently been proposed and has shown efficiency in achieving of results. In short, it detects non-critical checks (NCC) in the target program and removes them to explore new paths and analyze hard-to-reach vulnerabilities.

Unfortunately, these hybrid testing and patching-based fuzzing approaches still suffer from scaling to detect vulnerabilities that are hidden in deep execution paths of real-world applications. To be more specific, studies have shown that [24], the coverage of functions located within the abysmal program depths are quite hard to reach. Consequently, the hybrid testing approaches discussed here fail to detect a significant number of bugs entrenched within the software's depths, and patching-based fuzzers such as the T-Fuzz [23] will have a transformational explosion problem when the true bug is hidden deeply in the binary. To tackle these issues, we design and implement DeepDiver, a novel transformational hybrid fuzzing method. We expect to increase the code coverage as well as bug detection capabilities of the hybrid fuzzer by negating roadblock checks in target programs.

Without a doubt, disabling certain roadblock checks may break the original software logic, and vulnerabilities detected in the negated RC program might include *false positives* [25]. To filter out false positives, we implement a *bug validator* based on the post-processing symbolic execution that enables us replicate true bugs contained in the original software.

To demonstrate how efficiently the new transformational hybrid testing approach performs in comparison to the most modern methods, we evaluated DeepDiver on LAVA-M [26] dataset programs that displayed vulnerability, as well as on eight real-world software applications. DeepDiver discovered vulnerabilities in LAVA-M dataset in three hours whereas QSYM detected threats in five hours. Moreover, our tool outperformed all existing fuzzers like T-Fuzz [23] and AFLFast [27] in the presence of hard input checks and achieved significantly better code coverage.

The essential contributions of this paper are summarized as follows:

- We propose a set of novel approaches to aimed at improving the performance of our hybrid testing system. These approaches detect the fuzzing roadblock check and transform the target binary, including (a) analyzing checks of the target program in order to find a roadblock check that causes stoppage of fuzzing techniques, and (b) negating detected roadblock checks.

- To enhance hybrid fuzzing, we design DeepDiver with a bug validator. Bug validator determines whether a detected code weakness is a true bug or not and certifies the security strength by filtering false positives.

- We propose splitting floating-point comparisons into a series of the sign, exponent and mantissa comparisons for improving hybrid fuzzer capability if floating-point instructions are in the target program.

- We demonstrate the effectiveness of DeepDiver by testing LAVA-M dataset and eight real-world software applications. Comprehensive evaluation results show that our implementation can scale to various sets of real-world programs.

References

1. Mu, D.; Cuevas, A.; Yang, L.; Hu, H.; Xing, X.; Mao, B.; Wang, G. Understanding the reproducibility of crowd-reported security vulnerabilities. In Proceedings of the 27th USENIX Conference on Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 919–936.
2. Zalewski, M. American Fuzzy Lop (AFL), README. Available online: http://icamtuf.coredump.cx/afl/ (accessed on 29 October 2019).
3. Hocevar, S. zzuf:multi-purpose Fuzzer. Available online: http://caca.zoy.org/wiki/zzuf (accessed on 9 November 2019).
4. P. Oehlert; Violating Assumptions with Fuzzing. *IEEE Security & Privacy* **2005**, *3*, 58-62, 10.1109/msp.2005.55.
5. T-Fuzz Source Code. Available online: https://github.com/HexHive/T-Fuzz (accessed on 9 November 2019).
6. Month of Kernel Bugs. Available online: https://jon.oberheide.org/mokb/ (accessed on 10 December 2019).
7. Month of Browser Bugs. Available online: http://browserfun.blogspot.com (accessed on 10 December 2019).
8. The Heartbleed Bug. Available online: http://heartbleed.com/ (accessed on 10 December 2019).
9. Serebryany, K. OSS-Fuzz—Google's Continuous Fuzzing Service for Open-Source Software. Available online: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany (accessed on 10 December 2019).
10. Takanen, A.; DeMott, J.; Miller, C.; Kettunen, A. Fuzzing for Security Testing and Quality Assurance, 2nd ed.; Artech House: London, UK, 2018; pp. 1–2. ISBN 9781608078509.
11. Sutton, M.; Greene, A.; Amini, P. Fuzzing limitations and expectations. In Fuzzing: Brute Force Vulnerability Discovery; Addison-Wesley Professional: Crawfordsville, IN, USA, 2007; pp. 29–32. ISBN 0321446119.
12. Serebryany, K. LibFuzzer–a Library for Coverage-Guided Fuzz Testing. Available online: http://llvm.org/docs/LibFuzzer.html (accessed on 9 November 2019).
13. Fuzzing Tutorial: What is, Types, Tools and Example. Available online: https://www.guru99.com/fuzz-testing.html (accessed on 13 November 2019).
14. Pingfan Kong; Yi Li; Xiaohong Chen; Jun Sun; Meng Sun; Jingyi Wang; Towards Concolic Testing for Hybrid Systems. **2016**, , , .
15. Majumdar, R.; Sen, K. Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 416–426.
16. Yuekang Li; Bihuan Chen; Mahinthan Chandramohan; Shang-Wei Lin; Yang Liu; Alwen Tiu; Steelix: program-state based binary fuzzing. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017* **2017**, , 627-637, 10.1145/3106237.3106295.
17. Pak, B.S. Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. Master's Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2012.
18. Nick Stephens; John Grosen; Christopher Salls; Andrew Dutcher; Ruoyu Wang; Jacopo Corbetta; Yan Shoshitaishvili; Christopher Kruegel; Giovanni Vigna; Srdjan Capkun; et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *Proceedings 2016 Network and Distributed System Security Symposium* **2016**, , , 10.14722/ndss.2016.23368.
19. Koushik Sen; Gul Agha; CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. *Computer Vision* **2006**, *4144*, 419-423, 10.1007/11817963_38.
20. Cadar, C.; Sen, K. Symbolic execution for software testing: Three decades later. Commun. ACM 2013, 56, 82–90.
21. Xinyu Wang; Jun Sun; Zhenbang Chen; Peixin Zhang; Jingyi Wang; Yun Lin; Towards optimal concolic testing. *Proceedings of the 40th International Conference on Software Engineering - ICSE '18* **2018**, , 291-302, 10.1145/3180155.3180177.
22. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Conference on Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
23. Hui Peng; Yan Shoshitaishvili; Mathias Payer; T-Fuzz: Fuzzing by Program Transformation. *2018 IEEE Symposium on Security and Privacy (SP)* **2018**, , 697-710, 10.1109/sp.2018.00056.
24. Ognawala, S.; Hutzelmann, T.; Psallida, E.; Pretschner, A. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pau, France, 9–13 April 2018.

25. Cybersecurity 101: What You Need to Know about False Positives and False Negatives. Available online: https://www.infocyte.com/blog/2019/02/16/cybersecurity-101-what-you-need-to-know-about-false-positives-and-false-negatives/ (accessed on 3 November 2019).
26. Brendan Dolan-Gavitt; Patrick Hulin; Engin Kirda; Tim Leek; Andrea Mambretti; Wil Robertson; Frederick Ulrich; Ryan Whelan; LAVA: Large-Scale Automated Vulnerability Addition. *2016 IEEE Symposium on Security and Privacy (SP)* **2016**, , 110-121, 10.1109/sp.2016.15.
27. Marcel Böhme; Van-Thuan Pham; Abhik Roychoudhury; Coverage-based Greybox Fuzzing as Markov Chain. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16* **2016**, , 1032-1043, 10.1145/2976749.2978428.
