Integrating Directed-Based Fuzzing with AFL++ in QEMU Mode

Jin-myung Choi[1], Hyunjun Kim[1], Martin Kayondo[1], Yun-heung Paek[1]
[1]Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center(ISRC), [1]Seoul National University
{jmchoi,hjkim}@sor.snu.ac.kr, ,martin@snu.ac.kr, ypaek@snu.ac.kr

QEMU 모드에서 AFL++와 Directed-Based Fuzzing 의 통합

최진명[1], 김현준[1], 마틴[1], 백윤흥[1]
[1]서울대학교 전기정보공학부, [1]서울대학교 반도체 공동연구소

Abstract
Fuzzing is widely used as a testing tool to identify vulnerabilities in software programs. Although AFL++ has emerged to facilitate the integration and development of many fuzzers, there are still numerous advance fuzzing technologies that have not yet been incorporated. Among these, we have integrated state-of-the-art directed-based fuzzing techniques into AFL++ to operate in QEMU mode.

1. Introduction

With the rise in software complexity, there has been a noticeable increase in memory bugs that can lead to vulnerabilities that leak secret information, posing significant challenges for developers. As a result, researchers have led the way in creating effective methods to find and identify issues in software. Out of numerous mechanisms that aim to defend against software vulnerabilities, Fuzzing[1] stands out for its efficiency and effectiveness in discovering previously unfound vulnerabilities by generating a massive number of random inputs to run a target program in hopes of identifying certain input that triggers memory bug - i.e., program crash thus leading to various fuzzing research endeavors.
development of numerous fuzzers has branched into several categories based on target being fuzzed, such as the level of insight into the program being tested - white,gray or black box fuzzing or depending on how to generate and mutate inputs - mutation,generation, coverage based fuzzing, and having access to source code versus relying on executable binaries.
Unfortunately, non-trivial development efforts for both researchers and industry to make use of a combination of existing fuzzers for practical analysis or evaluate new tools. As AFL research progressed and AFL gained prominence, many fuzzers were developed based on AFL. However, because these were developed out-of-tree, it became difficult for both researchers and industry to make use of a combination of existing fuzzers for practical analysis or evaluate new tools. To address these issues, AFL++[2], emerged as an integrated framework for various fuzzers. AFL++ has introduced several improvements and new features that significantly enhance its efficacy and flexibility in software testing. These advancements include advanced mutators, optimized algorithms for faster execution, and more effective code coverage techniques. Moreover, AFL++ has extended its reach by supporting a variety of programming languages and platforms, broadening its applicability across a diverse range of software testing scenarios
Regrettably, despite receiving considerable attention for their performance, state-of-the-art

fuzzers, particularly those based on directed fuzzing such as AFLGO[3] and SelectFuzz[4], have not been integrated into AFL++. In this paper, we port and evaluate directed fuzzing techniques to AFL++. This work is non-trivial due to differing version of software stack (e.g. QEMU) and introduction of various APIs to manage and execute fuzzers. To that end, we implement AFLGo's and SelectFuzz's directed fuzzing algorithm that are source code based techniques and adapt them to AFL++'s binary fuzzer by modifying AFL++'s QEMU virtual machine. We evaluate and test our ported version of AFL++ on five programs with known bugs, and compare our implementation with AFL++'s coverage fuzzer as the control group. Our experimental results confirm the correctness of our implementation and the superiority of directed fuzzing over coverage fuzzing as previously reported in original AFLGo and SelectFuzz. Therefore, our work enhances the versatility and adaptability of AFL++, demonstrating its effectiveness as a comprehensive platform for deploying and evaluating advanced fuzzing techniques.

2. Background

2.1 coverage based, directed based

Coverage-based fuzzing is a method where a tool automatically generates inputs for a program to maximize the coverage of the program's code. It monitors which parts of the code are executed with each input and prefers inputs that explore new areas of the code. The aim is to find inputs that cause the program to behave unexpectedly, revealing potential bugs. Inputs that lead to unexplored code paths are considered valuable and are used to create more inputs.

Directed based fuzzing is a smart way to test software by focusing on specific areas that might have bugs, like recent changes or areas flagged by other tools. Unlike traditional methods that check the entire software, directed fuzzing zeroes in on these key spots to find issues more quickly and efficiently. It does this in three main steps: setting up by figuring out how far each part of the software is from the target area, the actual fuzzing where it picks and tweaks tests to get closer to the target, and finally, sorting through the results to find potential problems. This approach is especially good for analyzing complex software to uncover new problems or reproduce known issues, making it a powerful tool for improving software safety.

2.2 AFL++

American Fuzzy Lop++(AFL++), a renowned coverage-guided gray box fuzzer developed by Zalewski, mutates a set of inputs to reach previously unexplored path in the program. AFL's coverage guide evaluates input by monitoring how many different paths through the software's code are triggered and how frequently each path is taken during a single test. This is done by assigning each path a category based on the number of times it's executed, using groups that double in size (like 1, 2, 4, 8, etc.) to keep the amount of information manageable. If an input discovers a new category for a path, it's marked as valuable and kept for further testing. This data is recorded in a special table, with each section corresponding to a path. However, there's a limit to the table's size, which means some data might get mixed up. To handle this, AFL++ uses a smart method to select a core collection of tests that best represent the software's behavior, considering how fast and small these tests are.

When AFL++ has access to the source code, it employs static instrumentation to append metadata to basic blocks, thereby gathering the necessary information for coverage fuzzing. In contrast, when only the executable binary is available, AFL++ resorts to dynamic instrumentation through the use of a QEMU virtual machine. As the binary is executed within QEMU, the generated information is stored in shared memory, allowing AFL++ to access and continuously run tests. AFL++ places interesting inputs into the execution queue and runs the program again with slightly modified inputs through a mutator.

Whether an input is discarded or added to the execution queue depends on how interesting it is. Users can assign higher scores to inputs based on criteria they consider important, ensuring those inputs remain in the queue longer.

3. Design

In this section, we will explain overview and detail of design.

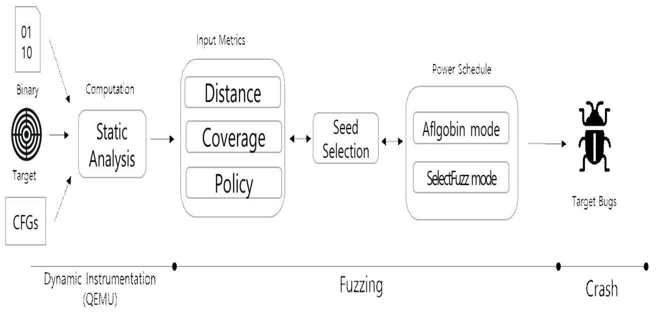

Figure 1: Overview of Fuzzing

We have modified the technologies originally based on source code, AFLGo and SelectFuzz, to work in QEMU mode. We are preparing the information needed to run them through dynamic instrumentation

To implement directed fuzzing, it's essential to acquire basic block addresses and distances between basic block and target function. This is achieved by employing analysis techniques such as static analysis[5] and concolic execution[6]. The derived addresses and distance information are stored in a file that is read by QEMU upon startup. After startup, This data is then stored in a one-dimensional array. The indexing for this array is done similarly to how AFL++ calculates hashes for bitmap management.

$$Array[((bb_{addr} \gg 4)\wedge(bb_{addr} \ll 8))\ \&\ (MAP\_SIZE-1)] = distance_{bb\ to\ target\ func}$$

Following the initial setup, dynamic instrumentation is performed via the QEMU virtual machine. This process involves instrumenting every basic block traversed for a given input. During this phase, in addition to the already captured coverage information, the system is designed to record the distance of each basic block to the target function. These distances are computed and aggregated to form a cumulative distance metric for each input. This aggregate distance is then stored in shared memory, ready for the fuzzer to access and utilize. The fuzzer then accesses this shared memory to evaluate the significance of the input based on the distance information, assigning scores accordingly. These scores determine whether an input is retained or discarded, a decision-making process known as power scheduling. The power scheduler employs an annealing-based Power Schedules approach to prioritize inputs that bring the execution closer to the target function, ensuring they remain in the execution queue.

4. Evaluation

Here are the experimental results comparing AFL++, AFLGO++, and SelectFuzz++, all implemented in qemu mode for afl++. The first column shows the total

Name	AFL++		AFLGO++		SelectFuzz++	
CVE-2018-8807 (decompileCALLFUNCTION)	115k (722)	0	52.3k (610)	3 (8.3h)	5.9k (441)	3 (63h)
mjs-issue-73 (mjs_mk_string)	1.6k (61)	0	40k (88)	9 (6.5m)	31k (81)	0
CVE-2018-20427 (getInt)	16.6k (440)	6 (21.9m)	134k (690)	7 (51.1m)	137k (723)	3 (2.27h)
CVE-2016-9827 (_iprintf)	12.5k (259)	0	14.9k (287)	0	8.6k (203)	0
CVE-2017-7578 (parseSWF_RGBA)	39.5k (660)	0	48.1k (674)	0	20.3k (574)	0

Table 1: Overview of out experiments

number of crashes and (in parentheses) the number of unique crashes, while the second column indicates how many vulnerability bugs we aimed to find were triggered, with the time of the first discovery noted in parentheses. Experiments were halted after a maximum of 120 hours if no vulnerabilities were found.

For CVE-2018-8807, while afl++ failed to find the vulnerability even after 120 hours, both aflgo++

and selectfuzz++ succeeded in discovering the desired vulnerability. This indicates that although afl++ is based on coverage-based fuzzing and shows a higher total number of crashes, directed-based fuzzing can also result in a high number of crashes if the targeted parts of the code are particularly vulnerable. Cases like CVE-2018-20427, 2016-9827, and 2017-7578 target functions related to decompiling, which involve many allocate and deallocate functions, leading to relatively more memory-related bug crashes. Despite the high number of total crashes, these were not effective in finding the targeted bugs because the fuzzing process prioritizes the inputs that caused crashes, making slight modifications to them for retesting. In such scenarios, directed-based fuzzing does not show good results. Moreover, aflgobin++ demonstrates superior performance over selectfuzz++ in terms of finding the target bug and the speed of discovery. This superiority is likely due to selectfuzz's limitations in handling data flow dependencies and indirect calls, which are necessary for computing distance information, particularly in QEMU mode.

5. Conclusion

While coverage-based fuzzing has been extensively developed for finding vulnerabilities in programs, if the binary to be tested already includes functions known to be vulnerable, using directed-based fuzzing is more efficient for quickly identifying if these functions are vulnerable. Therefore, it is significant that state-of-the-art coverage-based fuzzing techniques can be used in afl++, and that it can operate in QEMU mode even without source code.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277326), Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01840,Analysis on technique of accessing and acquiring user data in smartphone), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2024, Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government(MSIT), Inter-University Semiconductor Research Center (ISRC)

Reference

[1] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering, 47(11), 2312-2331.

[2] Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M. (2020). {AFL++}: Combining incremental steps of fuzzing research. In 14th USENIX Workshop on Offensive Technologies (WOOT 20)

[3] Böhme, M., Pham, V. T., Nguyen, M. D., & Roychoudhury, A. (2017, October). Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (pp. 2329-2344).

[4] Luo, C., Meng, W., & Li, P. (2023, May). Selectfuzz: Efficient directed fuzzing with selective path exploration. In 2023 IEEE Symposium on Security and Privacy (SP) (pp. 2693-2707). IEEE.

[5] F. Dong, C. Dong, Y. Zhang, and T. Lin, “Binary-oriented hybrid fuzz
testing,” in International Conference on Software Engineering and Service Science, 2015.

[6] J. Peng, F. Li, B. Liu, L. Xu, B. Liu, K. Chen, and W. Huo, “1dvul: Discovering 1-day vulnerabilities through binary patches,” in Proceedings of the 2019 International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, Jun. 2019.