ISBN: 978-981-94-1156-6
2024 the 14th International Workshop on Computer Science and Engineering (WCSE 2024)
doi: 10.18178/wcse.2024.06.041

MPFuzz: Dynamically Customizing Power Schedule by Using Seed Mutation Potential

Ben Wang [1] +, Xuan Zhang [1], Xiaoqi Zhao [2] and Yaojun Hou [1]

[1] College of Information Science and Engineering, Ocean University of China, Qingdao, China
[2] College of Information and Control Engineering, Qingdao University of Technology, Qingdao, China

Abstract. The widespread application of computer technology not only enhances work efficiency but also amplifies the threat posed by software vulnerabilities to personal privacy and social stability. In the field of automated vulnerability detection, Coverage-based greybox fuzzing (CGF) is highly esteemed for its excellent vulnerability discovery capabilities and wide applicability. However, the power schedule mechanism in existing CGF fuzzer has its limitations, particularly in terms of dynamics. It is difficult to effectively analyze and evaluate the real-time changes in the seed states during the fuzzing process, lacking sufficient information to assess the seeds' potential effectiveness and value. We propose a power schedule strategy based on mutation potential. This strategy involves evaluating the mutation value of seeds through real-time analysis of their performance across different execution sequences. Based on this evaluation, it adaptively allocates power, thereby reducing the waste of resources on low-quality seeds. We implemented a prototype MPFuzz and compared it against three other state-of-the-art fuzzers. The evaluation results show that MPFuzz exhibits better performance than other fuzzers in terms of code coverage.

Keywords: fuzzing, vulnerability detection, power schedule

1. Introduction

With the explosive growth in the number and functionality of application software, the complexity of code logic and system design is also increasing. The exploitation of software vulnerabilities has become a common method for attacking government websites and critical infrastructure services. In the current increasingly severe security environment, it is crucial to discover and repair software vulnerabilities in advance. Therefore, the development of efficient vulnerability detection technologies and related tools has become a research hotspot in the field of software engineering.

Fuzzing has a high degree of automation and can better handle complex and diverse programs. It is one of the most successful technologies in the field of vulnerability detection [1]. Its core is to input malformed data into the program under test, and then monitor whether the program has errors or crashes. Coverage-based greybox fuzzing leverages program feedback information collected by instrumentation to guide mutations towards unexplored program branches, significantly enhancing program coverage and vulnerability detection efficiency. Since CGF was proposed, security researchers have developed a series of advanced fuzzers such as AFL [2] and AFLFast [3], and successfully discovered tens of thousands of real software vulnerabilities.

Despite the significant achievements of CGF technology, it still has certain limitations. The current power allocation mechanism primarily relies on metrics such as the length and execution time of seeds to determine the number of mutations. It fails to provide sufficient information to accurately assess the mutation potential of seeds, and thus cannot effectively focus resources on seeds that are more likely to discover new paths.

This paper proposes a power allocation mechanism based on mutation potential. The key idea is to track and record the correlation between input fields and path branch instructions, categorize the seeds into

+ Corresponding author. Tel.: +8617864211363
E-mail address: 21210211169@stu.ouc.edu.cn.

different execution sequences, and allocate power based on the mutation value of the seeds. This ensures that fuzzing is more focused on high-quality seeds, enabling a deeper exploration of more complex paths.

We summarize our contributions as follows.

- We first propose a novel energy allocation mechanism based on mutation potential to improve the path exploration performance of CGF.
- We implement a prototype of MPFuzz.
- We evaluate MPFuzz with 3 state-of-the-art fuzzers on 8 open source software, and the experimental results show that MPFuzz outperforms other fuzzers.

2. Background

2.1. Overview of CGF

CGF usually maintains an infinite loop, picking a seed in each loop and mutating it to generate test cases. If a new program state is generated, the test case will be regarded as an interesting seed. This solution can effectively guide the fuzzer to explore more code coverage. Next, we will introduce the basic framework and related technologies of CGF.

The architecture of CGF is shown in Fig. 1. Initially, the source program undergoes instrumentation to collect edge coverage information during execution. Users must provide a series of initial seeds to enter the seed queue. Subsequently, a seed is selected through the seed scheduling to enter the power allocation stage. After being allocated a specific number of mutations, new test cases generated using the mutation strategy are executed on the target program, and the program's running state is monitored.

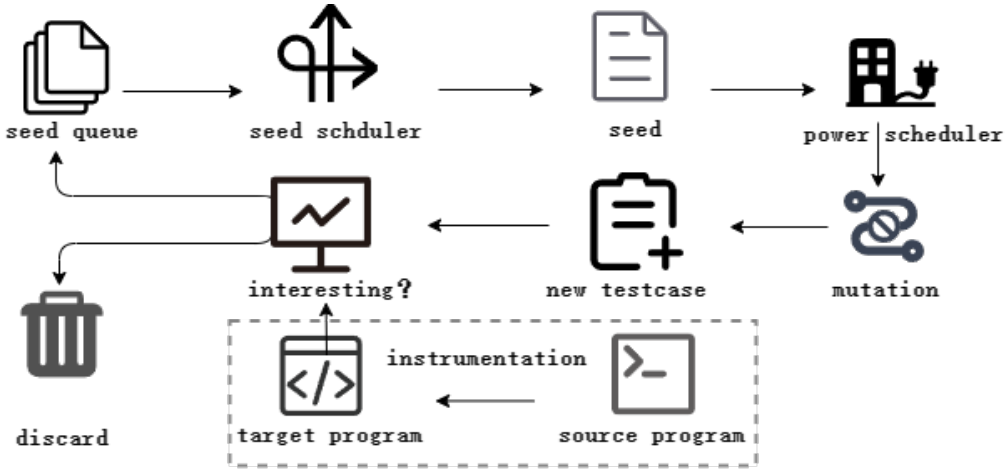

Fig. 1: Architecture of CGF.

CGF tracks changes in edge coverage information through basic block instrumentation and shared memory mechanisms. The seeds that generate new edge coverage will be added to the seed queue and given more resources, otherwise they will be discarded. CGF explores more new paths through new edge coverage guidance.

2.2. Limitations

Although researchers have conducted in-depth research and made significant progress in instrumentation feedback and mutation strategies. The energy allocation mechanism is still relatively blind, and existing CGF tools tend to allocate more energy to seeds with shorter execution times, ignoring the mutation value of seeds.

Take the program ImageMagick that converts PNG format to JPG format as an example. When we maliciously modify the magic number field of PNG sample A into a malformed data B in an invalid format, and then run ImageMagick with both seeds A and B for an equal duration, it is observed that seed B executes at triple the speed of seed A. However, the bulk of the test cases generated by malformed seed B mostly stay in the format validation area of ImageMagick, making it difficult to touch the deep branches of the program.

2.3. Related work

Existing CGF tools aim to reduce the input search space, for instance, by intelligently selecting mutation locations and specific mutation actions through program analysis or AI techniques [4-7][9]. Such refined strategies effectively utilize each mutation opportunity for more efficient resource utilization. However, the power allocation mechanism directly determines the number of mutations allocated to seeds, and allocating appropriate power to seeds of different quality can significantly speed up vulnerability detection.

AFLFast [3] prioritizes seeds, allocating computing resources to seeds that execute low-frequency paths. EcoFuzz [8] models the power scheduling strategy of AFL as an adversarial multi-armed bandit variant model, using an exploration-exploitation model to allocate power to different state seeds. MobFuzz [5] comprehensively considers multiple optimization objectives such as execution speed, memory consumption, and deep nested branches to achieve more efficient energy allocation. AIFORE [9] proposes a power schedule algorithm based on dynamic format inference. This algorithm determines whether the input format of new test cases needs to be reanalysed based on the coverage differences between test cases, and prioritizes allocating more power to seeds that are tested less frequently.

3. Methodology of MPFuzz

In order to improve the code coverage of CGF, we proposed MPFuzz, which uses instrumentation to track the comparison instruction sequence and extracts the execution sequence by analysing the correlation degree between the input field and the comparison instruction. Then, it evaluates the potential mutation ability of the seed based on the exploitation degree of the execution sequence to which it belongs. In addition, MPFuzz also considers the number of new edges discovered by a seed during the testing phase and compares it with the overall efficiency of discovering new edges for all seeds to measure the actual mutation ability of the seed. Finally, we develop a power schedule strategy based on the mutation potential of seeds.

3.1. Input-branch Dependency Analysis

Some fields of the input directly or indirectly affect branch constraints of the target program. Therefore, it is crucial to identify the degree of correlation between path branches and inputs. We designed an input-branch dependency analysis mechanism.

We monitor branch instructions (e.g., cmp, test and etc.) through instrumentation. Then, each byte of the seed is flipped in turn, and the correlation between the input and the program branch is determined by analysing the branch instruction sequence and the changes in the operands. Specifically, if a byte mutation causes the first operand of the comparison instruction to change, and the operand directly or indirectly corresponds to a part of the input, and the second operand remains unchanged, we analyse that there is a strong correlation between this input part and the path branch.

Fig. 2 illustrates the dependency analysis process on the second byte of input. After flipping the bytes and obtaining feedback information, we analyse that the second byte has a strong correlation with the comparison instruction. After all bytes are flipped, we define all branch instructions with strong correlation as execution sequences.

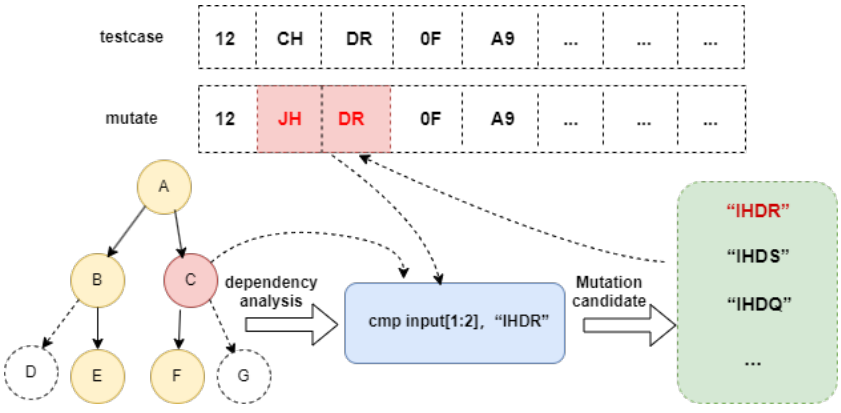

Fig. 2: Analyse which branch depends on the 2th byte.

Furthermore, if we want to explore basic block G, it is necessary to replace the second and third bytes with "IHDR". After analysing the dependencies between the second and third bytes and the branch, we simply modified the second operand "IHDR" of cmp to build a mutation candidate set, which will guide the refined mutation of the seed, faster explore unknown areas.

3.2. Power Schedule Mechanism

A seed with a large number of unexplored regions around an execution path has a greater potential mutation value, while a seed with more new branch coverage discovered in fewer executions has a greater actual mutation value. The power allocation mechanism of AFL does not fully consider the mutation potential of seeds. Based on AFL, we introduce two key factors, the potential effectiveness and actual effectiveness of seeds, to optimize the power schedule process.

As shown in Equation 1, E_s represents the energy allocated by MPFuzz. we define the ratio of the number of unexplored branch paths to the number of explored branches in the execution sequence to which the seed belongs as the seed potential effectiveness p_s. Within a certain time interval, the ratio of the number of new edges generated by the seed to the number of executions is determined as the actual effectiveness ev_s of the seed, and the ratio of the number of new edges to the total number of executions during this period is defined as the average effectiveness ev_{ave}. E_{afl} represents the original energy score of AFL.

$$E_s = E_{afl} \times \left(p_s + ev_s / ev_{ave} \right) \tag{1}$$

After determining the effectiveness of seeds, we introduce a schedule algorithm based on mutation potential. The algorithm traverses each seed s in the seed queue Q. For the initial seeds, we perform correlation analysis. In the non-deterministic stage, the algorithm combines the power allocation mechanism of AFL to perform precise power schedule based on the effectiveness of the seeds. In addition, analysing each seed consumes a lot of computing resources. Therefore, this paper uses a heuristic method to selectively extract valuable seed execution sequences, that is, analyse those seeds that can trigger new execution sequences.

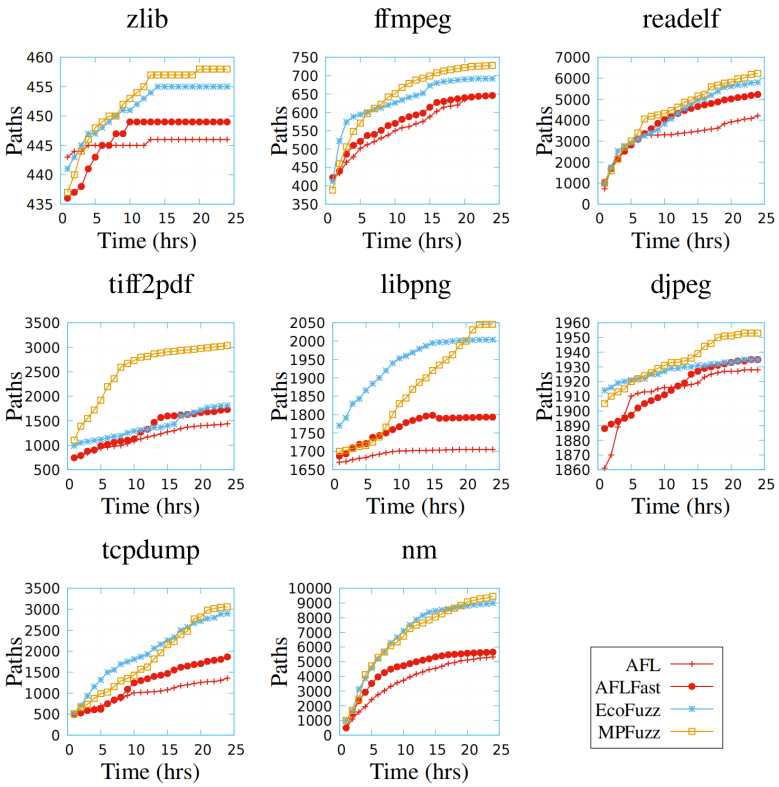

Fig. 3: The average branch coverage of different fuzzers.

4. Evaluation

In order to evaluate the effectiveness, we implemented the prototype MPFuzz. Specifically, we repeated each fuzzer 10 times to ensure the reliability of the data, each ran for 24 hours. Our measurements were taken on a system running Ubuntu 20.04.2 LTS using an Intel(R) Xeon(R) Gold 5218R CPU.

Benchmark Fuzzers. We select 3 fuzzers to evaluate our approach, including AFL, AFLFast, and EcoFuzz. AFL is one of the most commonly used Coverage-based greybox fuzzers. As mentioned before, ALFFast and EcoFuzz respectively optimize the power schedule part, which is the same as our goal.

Target Programs. We select 8 real-world software to evaluate our approach, including *readelf*, *tiff2pdf*, *libpng*, *djpeg*, *tcpdump*, *nm*, *zlib*, and *ffmpeg*. These software require different input formats to better evaluate the effectiveness of our approach.

Overall, MPFuzz has better path exploration ability. As shown in Fig. 3, MPFuzz shows significant advantages over other fuzzers in *zlib*, *ffmpeg*, *tiff2pdf*, *libpng*, and *djpeg*. In addition, from the analysis of the growth trend of the number of branches, MPFuzz demonstrates excellent continuous exploration ability, which is due to the mutation potential evaluation scheme adopted by MPFuzz. With a deeper understanding of the program, it can analyze the mutation value of seeds based on the program state, thereby providing more accurate guidance for power schedule.

5. Discussion

Performance Overhead. MPFuzz adds branch instruction instrumentation based on coverage instrumentation to capture the comparison branch sequence of the program. This operation slows down fuzzing speed. However, experimental results confirm that the sacrifice is worth it.

Limitation. MPFuzz is implemented through source code instrumentation and is not suitable for binary programs that are not open source. Although AFL implements runtime instrumentation by customizing QEMU. However, adding the branch instrumentation operation will bring huge performance overhead.

6. Conclusion

We propose a new approach to enhance CGF, which first performs comparison instruction instrumentation on the target program. Then, each byte of the input is randomly reversed during fuzzing, tracking program branch instructions that are strongly correlated with the input and determining the execution sequence. Subsequently, the method evaluates the potential mutation value of the execution sequence to which the seed belongs and the actual mutation value of the seed during the test process, and allocates reasonable power according to the mutation potential of the seed. Based on this idea, we design and implement the MPFuzz prototype. Furthermore, our evaluation results on 8 target programs show that MPFuzz is indeed effective in exploring branch branches.

7. References

[1] X. Zhu, S. Wen, S. Camtepe, et al. Fuzzing: a survey for roadmap . *ACM Computing Surveys*. 2022, pp. 1-36.

[2] M. Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[3] M. Böhme, V. T. Pham, A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1032-1043.

[4] C. Aschermann, S. Schumilo, T. Blazytko. REDQUEEN: Fuzzing with Input-to-State Correspondence. *In NDSS*. 2019, Vol. 19, pp. 1-15.

[5] G.Zhang, P. Wang, T. Yue, et al. MobFuzz: Adaptive multiobjective optimization in graybox fuzzing. *In Proceedings of the Network and Distributed Systems Security Symposium*. 2022, pp. 1-18.

[6] A. Fioraldi, D. C. D'Elia, E. Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. *In Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 2020, pp. 1-13.

[7] K. Zhang, X. Xiao, X. Zhu. Path transitions tell more: optimizing fuzzing schedules via runtime program states. *In Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. 2022, pp. 1658-1668.

[8] T. Yue, P. Wang, Y. Tang, et al. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. *29th USENIX Security Symposium* 2020, pp. 2307-2324.

[9] J. Shi, Z. Wang, Z. Feng, et al. AIFORE: Smart fuzzing based on automatic input format reverse engineering. *32nd USENIX Security Symposium*. 2023, pp. 4967-4984.