ISBN: 978-981-18-7950-0
2023 the 13th International Workshop on Computer Science and Engineering (WCSE 2023)
doi: 10.18178/wcse.2023.06.005

DGFuzz: Dynamically Constructing Control Flow Graph to Guide Greybox Fuzzing

Wenjie Lv [1] [+], Xiaoqi Song [1], Cong Wang [2], Wenbin Zhang [2] and Haipeng Qu [1]

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

[2] Information and Telecommunications Company, State Grid Shandong Electric Power Company, Shandong, China

Abstract. In this era of information explosion, software with various functions is emerging and the harm that software vulnerabilities can cause is increasing. Coverage-based greybox fuzzing (CGF) is already known as one of the most effective software vulnerability detection methods available, due to its simple principle and excellent results. However, the mutator is blind to the exploration of undiscovered paths, it cannot know how to search the input space to explore new paths, and can only mutate all input bytes or random bytes in an attempt to search for new paths. In this paper, we propose a method based on the dynamic construction of control flow graph to guide the mutation direction of the fuzzing. Based on the explored control flow information, we can analyze where still unexplored basic blocks remain and combine with input branch dependency analysis to mutate toward unexplored paths. We implemented a prototype DGFuzz and benchmarked it against four other state-of-the-art fuzzers. The evaluation results show that DGFuzz can discover more new branches than other fuzzers.

Keywords: fuzzing, vulnerability detection, code coverage

1. Introduction

The security vulnerabilities have become the most significant threat in program, as software becomes more complex. Many vulnerabilities can be exploited by attackers to do something malicious, such as execute malicious code, privilege escalation attacks, etc. As the high threat of software vulnerabilities, finding as many vulnerabilities as possible in the software before it is released is what software engineers are currently trying to do. Recently, many approaches have been proposed to discover vulnerabilities, with many efforts from industry and academia.

Fuzzing is the most popular technique in recent years, which is an automated testing technique aims to find vulnerabilities. The key idea of fuzzing technique is to attempt to trigger irregular behaviours (e.g., crashes) in the program by generating a large amount of untrust input. Coverage-based greybox fuzzing (CGF) has become the commonly used approach, as its excellent effectiveness and efficiency. One of the most well-known state-of-the-art CGF fuzzer is American Fuzzy Loop (AFL) [1], and thousands of vulnerabilities have been found by AFL. It uses a compile-time instrumenting technique to capture the execution paths and coverage of the target program.

Although CGF has developed very rapidly over the years, but there are still many difficulties need to be solved. The program control flow is complex, and the fuzzer do not have enough knowledge to accurately make test cases mutate towards untriggered codes. Many works have focused on making the discovery of new path easier, but the fuzzer still has no way of knowing which branches are not triggered.

This paper introduces a new method guiding fuzzer to search for untriggered codes. The key idea is constructing control flow graph (CFG) during testing, guiding fuzzers to explore the missing basic blocks in the CFG. The main difficulty currently faced by fuzzing during mutation is that the mutation process is blind and it is difficult for fuzzers to know the relationship between the unexplored basic blocks and the explored code. Therefore, we propose a method based on constructing CFG dynamically to reveal the relationship

[+] Corresponding author. Tel.: +8617866623716
E-mail address: lwj3656@stu.ouc.edu.cn.

between explored and unexplored codes. Thus, the fuzzer knows which branch to focus on to explore the new code and thus locates the corresponding mutation location.

We summarize our contributions as follows.

- We first propose a lightweight generalized framework for guiding mutation, DGFuzz, to improve the path exploration performance of CGFs.
- We implement a prototype of DGFuzz.
- We evaluate DGFuzz with 4 state-of-the-art fuzzers on FuzzBench [2], and the experimental results indicate that DGFuzz outperforms all other fuzzers.

2. Background

2.1. Overview of CGF

Coverage-based Greybox Fuzzing (CGF), which tracks the execution path of the target program by compile-time instrumentation, continuously mutates the test cases to generate new test cases to explore the untriggered program branches of the target program. One of the most popular CGF fuzzers AFL, and the following we will introduce the architecture and principle of CGF with AFL as an example.

The architecture of AFL is shown in Fig. 1. AFL needs an initial corpus to build a seed queue, select one seed from the seed queue to the mutator to mutate, run target program with the mutated testcase, and observe the program behavior of the target program.

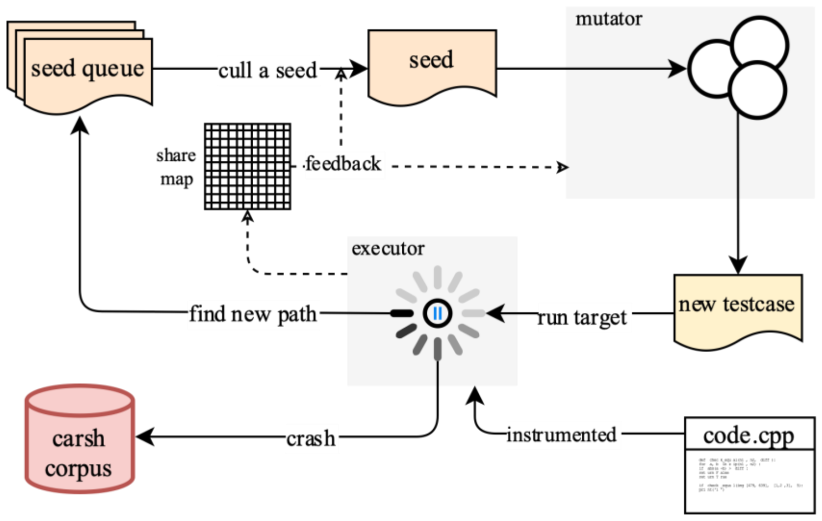


Fig. 1: Architecture of AFL

AFL keeps track of the target runtime state, mainly the edge coverage information, during the execution of the target program by instrumented code and shared memory. The selection and mutation of test cases are guided based on the overall edge coverage, and AFL wants to trigger code that has never been executed by mutation. Thus, the design of feedback is one of the most central modules in AFL.

2.2. Limitations

The feedback and mutation strategies of AFL, although excellent, are still relatively blind in terms of coverage exploration. AFL uses a violent search method to explore new branches, with no way to know the relationship between the input and each branch. For example, one file parsing program, the program will only read the first 16 bytes of the file, but the file size is 1KB, all the remaining bytes are useless for this program, but AFL still mutates them anyway. In addition, mutator lacks understanding of the program control flow graph.

2.3. Related work

Recently, many CGF fuzzers have been proposed to improve the coverage of the target inspired by AFL [3-6]. And reducing the search space of input during fuzzing is a hot topic of current research in the field of fuzzing. The key to these efforts is generally to better guide the mutation process by understanding the feedback information from the program.

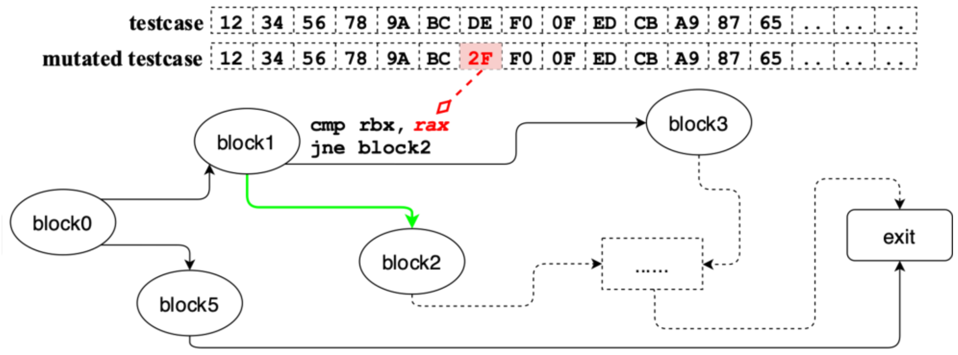
FairFuzz [7] optimizes the scheduler and mutator, which prefer to select a test case that hit rare branches and mutate the key position in the test case. Truzz [8] tries to guide the mutation process to avoid entering the error handling process branch, which is based on the feedback of the changes of path length. ProFuzzer [9] mutates the input byte by byte and then observes the program behavior to identify the semantic and field information of the input (e.g., Loop count, Size, Enumeration), thus guiding the mutator. RedQueen [10] finds that the input and current program status have a strong input-to-status correlation, as part of the input is passed directly into memory or registers for comparison. It can automatically overcome checksums and magic bytes through a lightweight implementation for input-to-status correspondence.

3. Methodology of DGFuzz

To enhance CFG's path exploration performance, we propose DGFuzz, which describes the coverage of the target program by the technique of dynamically constructing control flow graph. DGFuzz analyzes the dependency of input bytes with program branches through the feedback information of the program, and collects the control flow graph information of the program by instrumenting at the branches.

3.1. Byte dependency analysis

The control flow transfer of program branches is influenced by the data at the branch, so it is important to find out which input bytes affect each branch. We design a data flow feedback mechanism to analyze byte dependencies.


testcase
12 34 56 78 9A BC DE F0 0F ED CB A9 87 65 .. .. ..
mutated testcase
12 34 56 78 9A BC 2F F0 0F ED CB A9 87 65 .. .. ..
block1
block3
block0
block2
......
exit
block5

```
cmp rbx, rax
jne block2
```

Fig. 2: Analyze which branch depends on the 7th byte (DE).

We monitor the instructions that affect conditional jumps (e.g., *cmp*, *test*, *and*, etc.) and then flip each byte of the input in turn to get the branch dependency of each byte based on the data flow feedback. Fig. 2 explains the process of analyzing the dependency of the 7th byte of the input to the branch. If we want to explore another branch of block1, the 7th byte of input is which we need to mutate.

3.2. Dynamic construction of CFG

AFL's strategy does not have enough knowledge to understand which branches are not triggered so that effective guidance is not possible during the mutation stage. We designed a method to dynamically construct CFG during fuzzing to expose the unexplored branches. For example, if a conditional *jmp* instruction has only transferred to the same branch so far, it is obvious to know that there is still a branch that is not triggered (because the conditional *jmp* must correspond to two different branches), and based on the byte dependency analysis, a custom mutation can be performed for the unexplored branch.

We added an instrumentation module to collect the information needed to build the control flow graph, which is organized in a tree structure. The code snippet is instrumented into each basic block, which is used to record the id of the previous basic block. All the basic block that has been discovered are saved in a global CFG. If a new basic block is discovered, it needs to be added to the global CFG. Based on the number of child nodes in each basic block, we can target our mutators to perform specific mutations.

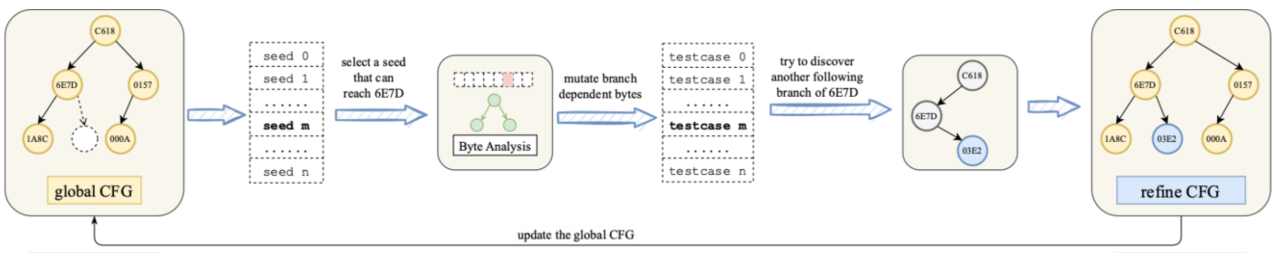


Fig. 3: Explore a new branch by dynamically constructing CFG.

Fig. 3 depicts the process of DGFuzz exploring a new branch. DGFuzz first identifies a conditional branch for which there is an opposing condition that has not been triggered, and finds a test case that can trigger that branch by byte dependency analysis. Then, the bytes in each test case that depend on that branch are mutated to try to explore the new branch to the opposing condition.

4. Evaluation

In order to evaluate the effectiveness of DGFuzz, we implemented a prototype and evaluate its effectiveness on the google open source FuzzBench [2] framework. Specifically, we ran each fuzzer for 24 hours, and then repeated the experiment for 5 times. Our measurements were performed on a system running Ubuntu 20.04.2 LTS with Intel(R) Xeon(R) Gold 5218R CPU.

Benchmark Fuzzers. We select 4 fuzzers to evaluate our approach, including AFL [1], ProFuzzer [9], AFLFast [11], and FairFuzz [7]. AFL is one of the most well-known CGF fuzzers. FairFuzz and ProFuzzer, as mentioned earlier, optimize the feedback information and mutators, the same as our goal.

Target Programs. We select 11 targets to evaluate our approach, including bloaty, freetype, harfbuzz, jsoncpp, libjpeg-turbo, libpng, mbedtls, php, re2, vorbis and woff2. Since they require different input formats, they can better represent the experimental effect.

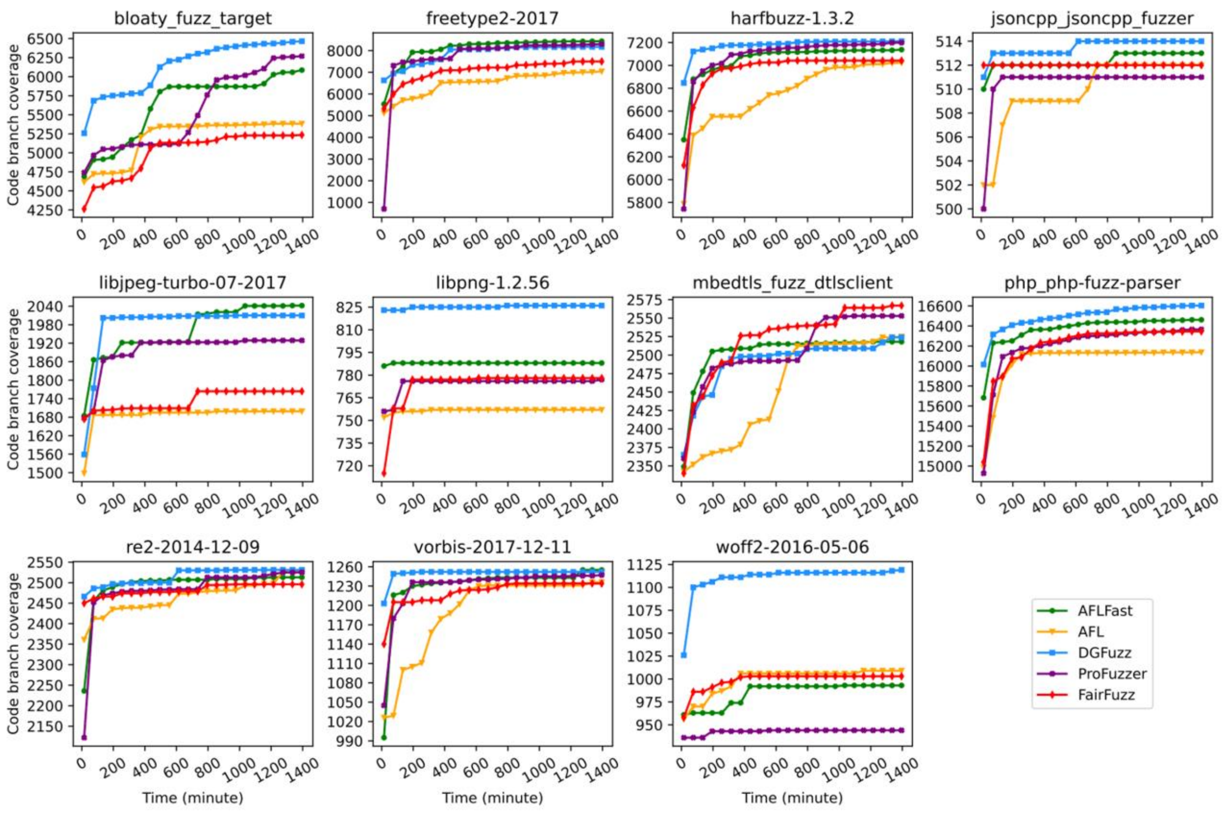


Fig. 4: The average branch coverage of different fuzzers for 5 runs of 24 hours.

Fig. 4 shows the result of our experiments. DGFuzz achieved very promising experimental results in 10 out of 11 targets, and was found to have slightly fewer branches explored than FairFuzz and ProFuzzer on the mbedtls target. This is because the logic of the mbedtls cryptographic library is highly complex, and the instrumentation module added by DGFuzz performed slightly worse on mbedtls, resulting in slightly slower testing speeds. Overall, compared to the other 4 fuzzers, DGFuzz has an outstanding performance in terms of code coverage.

5. Discussion

Overhead of execution speed. DGFuzz slows down the execution, because two instrumentation modules are added, in order to obtain the control flow and data information of the program. However, from the results of our experiments, this part of the overhead is acceptable.

Limitations. DGFuzz has limitations in analyzing input-branch relationships that undergo complex implicit propagation. If a program would first classify each byte of the input, and byte-flip one input byte before and after belonging to the same class, it will not be possible to analyze which branch is dependent on this byte.

6. Conclusion

We propose a new approach to enhance CGF, which uses techniques for dynamically constructing control flow graph during fuzzing. This can reduce the search space of the input and avoid making many meaningless mutations. Combining the two methods of byte dependency analysis and dynamic construction of control flow graph, the fuzzer can mine the unexplored basic block branches more precisely. Based on this idea, we designed and implemented a prototype DGFuzz. As well, our evaluation results on 8 target software show that DGFuzz is indeed very effective in enhancing coverage.

7. References

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

[2] J. Metzman, L. Szekeres, L. Simon. FuzzBench: an open fuzzer benchmarking platform and service. *In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2021, pp. 1393-1403.

[3] M. Böhme, V.T. Pham, M. Nguyen. Directed greybox fuzzing. *In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2017, pp. 2329-2344.

[4] V.T. Pham, M. Böhme, A.E. Santosa. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*. 2019, 47(9), 1980-1997.

[5] S. Nagy, M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. *In 2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 787-802.

[6] D. She, K. Pei, D. Epstein. Neuzz: Efficient fuzzing with neural program smoothing. *In 2019 IEEE Symposium on Security and Privacy (SP).* 2019, pp. 803-817.

[7] C. Lemieux, K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. *In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 2018, pp. 475-485.

[8] 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.

[9] W. You, X. Wang, S. Ma. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. *In 2019 IEEE symposium on security and privacy (SP).* 2019, pp. 769-786.

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

[11] 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.