Mutation Methods for Structured Input to Enhance Path Coverage of Fuzzers

Yonggon Park, Youngjoo Ko, and Jong Kim(✉)

Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang, South Korea
{nanimdo,y0108009,jkim}@postech.ac.kr

Abstract. Existing mutation methods used in coverage-based grey-box fuzzing (CGF), such as those employed by AFL and AFL++, can lead to biased testing for structured inputs. While fuzzing, certain input sections of structured input may receive fewer mutations, resulting in less testing of the code that handles those sections, which leads to lower path coverage in those code parts.

In this paper, we propose two mutation methods for the structured input to address the unbalanced problem and improve path coverage. The first method, Uniform Mutation, involves conducting additional mutations in input sections that trigger less testing, thereby achieving a more balanced path coverage across the target program. However, this method requires prior knowledge of the input format, which reduces its usability when the format of the target program changes. To overcome the limitation, we propose the second method, Format-agnostic Mutation, which automatically partitions the input into sections based on coverage feedback. This method redistributes the number of mutations and resizes the sections to improve path coverage without knowing the input format.

We evaluate the effectiveness of these methods using two real-world programs (Xpdf and libxml2) and compare them with AFL. The experimental results demonstrate that Uniform and Format-agnostic mutations (weight and resizing) outperform AFL regarding path coverage exploration.

Keywords: Fuzzing · Mutation · Structured Input · Path Coverage · AFL

1 Introduction

Fuzzing can be classified as black-box, white-box, or grey-box, depending on the level of awareness about the program structure. Grey-box fuzzing methods [5,6,23] employ lightweight instrumentation techniques to gather information about the program. This instrumentation introduces minimal overhead compared to the analysis techniques used in white-box fuzzing [7,9,10]. By leveraging the obtained information, grey-box fuzzing methods can generate inputs

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
H. Kim and J. Youn (Eds.): WISA 2023, LNCS 14402, pp. 257–268, 2024.
https://doi.org/10.1007/978-981-99-8024-6_20

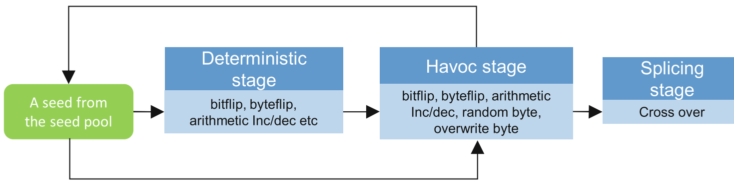

Fig. 1. The mutation method in AFL [23]

that are more effective at triggering bugs than inputs generated by black-box fuzzing [4,8,19].

Coverage-based grey-box fuzzing (CGF) is a widely used technique in the field of grey-box testing for detecting security vulnerabilities in real-world programs. CGF leverages metrics such as path or code coverage, acquired through lightweight instrumentation, to generate test inputs. Its primary objective is to enhance the chances of triggering vulnerabilities that may be present in less frequently executed or unexplored paths within the target program. Typically, CGF involves several key components, including seed selection, power schedule, mutation, execution, and seed evaluation with feedback. Initially, CGF selects a seed from a set of initial inputs and generates multiple test cases by applying mutations to the selected seed. The power schedule then determines the number of test cases created from each seed. Subsequently, CGF executes the program with the generated test cases and collects coverage information. If a test case executes previously unexplored code locations, CGF identifies it as a new seed to explore in the subsequent fuzzing iteration cycle. This iterative process enables CGF to systematically explore different paths of the program and increase the likelihood of uncovering vulnerabilities.

Representative fuzzers, such as AFL, AFL++, and VUzzer, employ the mutation method illustrated in Fig. 1, which involves modifying inputs from the beginning to the end while treating all input parts equally. This mutation process typically employs predetermined bitflips and additions as the modifying operations. The mutation methods employed by these fuzzers generally operate in three stages: deterministic, havoc, and splicing. In the deterministic stage, all input positions are mutated using the predetermined operations. In the havoc stage, positions for mutation are randomly selected from the input. Occasionally, an input is mutated through a crossover operation in the splicing stage. These stages collectively enable the fuzzers to systematically modify inputs and explore different paths, increasing the likelihood of triggering bugs or vulnerabilities in the target program.

However, we have observed that existing mutation methods exhibit a bias towards specific input sections, often neglecting others. This approach may not be optimal for programs that rely on structured inputs, where different input sections dictate the execution of different code segments. For example, consider the case of PDF file inputs, which consist of distinct sections such as the header, data, cross-reference table, and trailer (depicted in Fig. 2). Each section is pro-

cessed by different code segments within the program, making their comprehensive coverage essential. To evaluate the impact of existing mutation methods on path coverage exploration for PDF inputs, we conducted a 24-hour fuzzing experiment using 30 randomly selected PDF files. The distribution of mutations applied to each section during the fuzzing process is depicted in Fig. 3. The results revealed a significant disparity in the distribution of mutations. Approximately 90% of the mutations were applied to the data section, while the header and cross-reference table sections received less than 3% of the mutations. This observation indicates that the mutation process primarily focused on the larger data section, disregarding smaller sections like the header. Consequently, it fails to consider crucial information about the most effective locations in the input for effective fuzzing. Consequently, certain input parts, such as the header section or less frequently accessed sections, may have fewer mutations, potentially leading to lower path coverage in those areas. This biased mutation approach limits the exploration of specific regions within the input and may hinder the detection of vulnerabilities or bugs associated with those neglected sections.

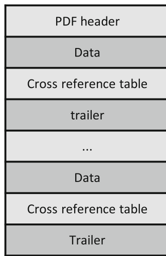

Fig. 2. The structure of PDF file

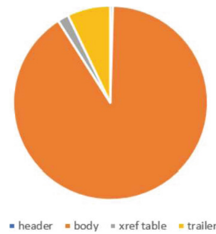

Fig. 3. Mutation rate of each section of PDF by AFL

We propose two primary mutation methods to address the problem and achieve high path coverage. The first method, Uniform mutation, tackles the issue by introducing additional mutations in input sections that have triggered fewer tests for a particular part of the target program. Its goal is to achieve a more balanced path coverage across the entire program. However, a prerequisite for utilizing this method is prior knowledge of the input format, which may limit its applicability. Furthermore, implementing this method requires extra effort whenever there are changes in the input format or the corresponding part of the target program. To overcome this limitation, we introduce the second method, Format-agnostic mutation. This method automatically divides the input into sections based on feedback obtained from coverage analysis. By partitioning the input, it redistributes the number of mutations and adjusts the sizes of sections to enhance path coverage. The Format-agnostic mutation method eliminates the need for explicit knowledge of the input format and ensures adaptability to changes in the input structure and target program.

The effectiveness of these methods is evaluated by measuring their performance on real-world programs (Xpdf and libxml2) and comparing them with AFL, a popular fuzzer. The evaluation clearly demonstrates that the proposed methods outperform AFL regarding path coverage exploration. This research offers the following key contributions:

- We introduce novel Uniform and Format-agnostic mutation methods.
- We demonstrate the efficiency of the Uniform mutation method in increasing path coverage.
- We effectively partition the input into sections, leading to higher path coverage compared to conventional fuzzers like AFL, while still maintaining usability.

2 Background

2.1 Coverage-Based Grey-Box Fuzzing

Coverage-based grey-box fuzzing has been widely used and detected many vulnerabilities in real-world programs. It generates testing inputs by leveraging lightweight instrumentation that extracts the coverage information, such as path and code coverage. The coverage information helps to explore the program's deep paths and detect bugs and vulnerabilities [1,5,6,11–13,15,21,23].

Grey-box fuzzing follows a typical workflow that includes seed selection, power schedule for energy assignment, seed mutation, execution feedback, and seed evaluation. Allow us to provide a brief description of the grey-box fuzzing workflow: The process begins with selecting a seed from the seed pool. The initial seed pool consists of regular inputs known as seeds. This seed selection process determines which seed from the pool is used to generate test cases, which serve as input for testing the target program. The power schedule plays a crucial role in determining the number of test cases, referred to as energy (E), that will be generated from the selected seed. The fuzzer applies mutation techniques to the seed based on the energy, creating new test cases. The target program executes each test case, which provides feedback during execution. This feedback typically includes coverage information, highlighting which code paths, branches, or functions were traversed. Leveraging this feedback, the fuzzer evaluates the input and identifies cases that increase coverage or exhibit abnormal behaviors. These interesting inputs are considered valuable and are added to the seed pool for further exploration in subsequent iterations. The fuzzing process continues iteratively, generating new test cases, executing them, and evaluating their impact based on the coverage feedback collected. Once all seeds in the seed pool are selected, the fuzzer selects seeds again from the beginning so they can be selected multiple times.

2.2 Mutation Method

We explain the mutation method used in AFL (shown in Fig. 1) because many coverage-based fuzzers have been implemented based on AFL and have adopted

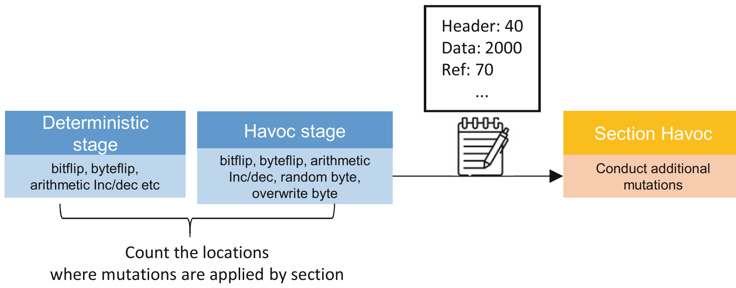

Fig. 4. The workflow of Uniform mutation. The yellow stage, section havoc, is the mutation stage that we newly added. (Color figure online)

a similar mutation method. This method consists of several stages, including deterministic, havoc, and occasionally splicing. Let us delve into each stage:

– **Deterministic stage.** AFL applies predetermined mutation operations to the input data in a systematic manner. These operations are typically performed on every bit or byte of the input. The deterministic stage encompasses mutation operators such as bit flips, byte flips, arithmetic increments/decrements, and other simple transformations.
– **Havoc stage.** In the havoc stage, randomness is introduced into the mutation process. AFL randomly selects positions by offsets within the input data and modifies the bytes or bits at those positions. These modifications can involve altering values, flipping bits, or applying arithmetic operations.
– **Splicing stage.** The splicing stage combines portions of two or more different inputs to generate new test cases. It is important to note that this stage is occasionally conducted.

While most mutations in AFL primarily occur in the deterministic and havoc stages, it is noteworthy that these mutation stages lack information regarding which positions in the input are particularly effective for fuzzing.

3 The Proposed Mutation Methods

3.1 Uniform Mutation

The first proposed mutation method, Uniform mutation, aims to tackle the inequality problem present in existing mutation methods by introducing additional mutations in input sections that have triggered less testing within the target program. Figure 4 illustrates the workflow of Uniform mutation with the newly added component, the section havoc stage. The fuzzer needs prior knowledge of the section structure of the input to facilitate fuzzing with the boundaries and divisions of different sections within the input. The mutation algorithm follows the standard execution of the deterministic and havoc stages. Upon completing the existing mutation stage, the fuzzer keeps track of the number of

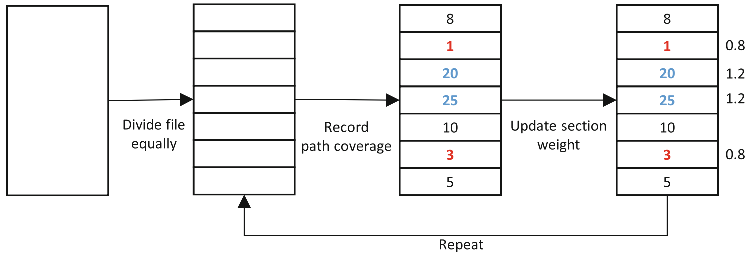

Fig. 5. Format-agnostic mutation with control of weight

mutations performed in each input section during the deterministic and havoc stages. This allows comparing the number of mutations across sections to identify those that have undergone less testing. To address the imbalance problem, we incorporate the section havoc stage into the fuzzer, wherein random positions within the sections requiring additional mutations are selected. Random offsets are chosen like the existing havoc stage, and additional mutations are applied at these positions. By integrating the section havoc stage into the mutation method, Uniform mutation ensures a more equitable distribution of mutations across input sections. This approach helps mitigate coverage imbalances and increases the likelihood of exploring new paths in the target program that have received less testing.

3.2 Format-Agnostic Mutation

The second mutation method, Format-agnostic mutation, aims to increase the usability of the prior Uniform mutation by allowing arbitrary division of the input into sections when there is no prior knowledge of the input format structure. We propose two versions of Format-agnostic mutation, as shown in Figs. 5 and 6.

The first version is the Format-agnostic mutation with weight, depicted in Fig. 5. This method divides the input into sections with equal size and weight, which might be not consistent with the actual section structure. Subsequently, it applies existing mutation methods to each section, according to the weight given to each section. Throughout this process, the fuzzer keeps track of the number of discovered paths for each section, serving it as a measure of path coverage. Based on this information, the fuzzer calculates a distinct weight for each section. The weight calculation is adjustable and determines the weight ratio using the following heuristic: Sections in the top one-third of path coverage receive a weight increase of 20%, while sections in the bottom one-third experience a weight decrease of 20%. We repeated above process 10 times, to form more precise and useful section information. In Fig. 5, the blue part corresponds to the top one-third, indicating a weight increase, while the red part corresponds to the bottom one-third, reflecting a weight decrease.

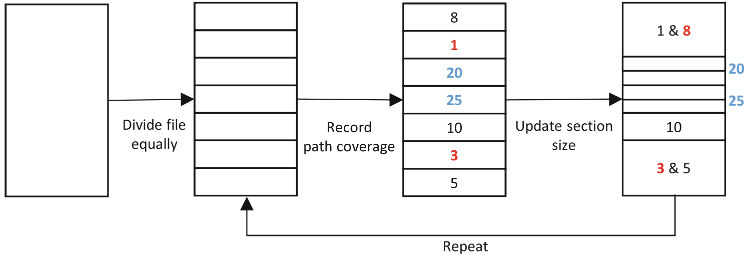

Fig. 6. Format-agnostic mutation with control of section size

The second version is the Format-agnostic mutation with resized sections, illustrated in Fig. 6. Similar to the previous version, it evenly divides the input into sections and applies existing mutation methods to each section. Each section's path coverage is evaluated like the Format-agnostic mutation with weight approach. Sections within the top one-third of path coverage undergo resizing by dividing them in half. Conversely, sections within the bottom one-third of path coverage are merged with adjacent sections exhibiting low path coverage. Following the resizing of sections, the fuzzer once again proceeds with the mutation algorithm, targeting the resized sections. In Fig. 6, the blue sections, which have discovered 20 and 25 paths, are divided, while the red sections, with only one and three paths, are merged with neighboring low-coverage sections.

4 Evaluation

Prototypes of the Uniform mutation and two versions of the Format-agnostic mutation (weight and resizing sections) are implemented on AFL as part of our research. The experiments are conducted using Xpdf and libxml2 as the target programs. The performance of the Uniform and Format-agnostic mutation methods was compared to that of AFL. The primary focus of the evaluation was to assess the path coverage achieved by each method. Our evaluation aims to address the following research questions:

RQ1. Does the Uniform mutation approach, which targets input sections with knowing the input structure, enhance the coverage exploration capabilities of fuzzing?

RQ2. Can we attain high path coverage by automatically dividing the input into sections without knowing the input structure?

By conducting comprehensive experiments and analyzing the results, we provide insightful answers to these research questions, shedding light on the benefits and potential of the Uniform and Format-agnostic mutation methods in improving coverage exploration during fuzzing.

4.1 Experiment Setup

All of our evaluations were performed on an AMD Ryzen 7 6800H with Radeon Graphics @ 3.20 GHz (4 MB cache) machine with 8 GB of RAM. The O.S. is Ubuntu 20.04 with Linux 5.15.0-72-generic 64-bit. We tested Xpdf and libxml2 for six hours.

4.2 The Result of Uniform Mutation

Table 1 shows the number of paths found by AFL and the Uniform mutation method. The experiment was conducted multiple times (five times) on the Xpdf benchmark for six hours to ensure fairness. The average results showed that the Uniform mutation method explored 9.64% more paths than AFL.

Table 1. The # of paths found by AFL and Uniform mutation on Xpdf.

Test Number	AFL (path)	Uniform (path)
#1	4011	4769
#2	4705	4754
#3	4083	4744
#4	4745	4722
#5	4052	4689
Avg	4319.2	4735.6

AFL's performance demonstrates inconsistency, whereas the Uniform mutation method consistently produces stable results. This inconsistency in AFL's performance can be attributed to the imbalance of mutations. AFL mutates the input by flipping all positions in the input one by one (deterministic stage) or randomly selecting positions with offsets to modify bytes or bits (havoc stage) without considering the input section. As a result, the code handling each input section is not tested with an equal chance. In contrast, the Uniform mutation method achieves stable results and higher path coverage by focusing on the exploration of smaller sections (such as the header and the cross-ref) based on the input section format. By uniformly applying mutations based on the input sections, this method achieves enhanced coverage and maintains stable fuzzing performance across multiple experimental attempts. These results clearly demonstrate the advantages of the Uniform mutation method over AFL in terms of path coverage and stability in fuzzing performance.

4.3 The Result of Format-Agnostic Mutation

Table 2 displays the path coverage results obtained from AFL, Uniform mutation, and the Format-agnostic mutations (weight and resize strategies) on the Xpdf

benchmark. The experiment was repeated five times, with each run lasting six hours. On average, the Format-agnostic mutations (weight and resize strategies) revealed 7.0% and 6.4% more paths, respectively, compared to AFL. However, they still exhibited lower path coverage when compared to the Uniform mutation method, which has prior knowledge of the input format. These findings highlight that while the format-agnostic mutations achieved some improvements in path coverage compared to AFL, the Uniform mutation approach, benefiting from its understanding of the input format, outperformed the other methods by achieving higher path coverage.

Table 2. The # of paths found by AFL and Format-agnostic mutation on Xpdf.

Test Number	AFL	Uniform mutation	Format-agnostic (weight)	Format-agnostic (resize)
#1	4011	4769	4660	4665
#2	4705	4754	4654	4517
#3	4083	4744	4659	4525
#4	4745	4722	4567	4629
#5	4052	4689	4559	4647
Avg	4319.2	4735.6	4619.8	4596.6

The Format-agnostic mutation method is proposed as a solution for cases where prior knowledge of the input structure is unavailable. We observed that this method effectively divides the input into sections, resulting in only a slight difference in performance compared to the Uniform mutation method. When comparing the two versions (weight and resize strategies) of the Format-agnostic mutation method applied to the PDF input, the weight strategy shows similar performance to the resize strategy. This can be attributed to the PDF input's simple section structure, which aligns well with the divided sections determined by the Format-agnostic mutation method. Furthermore, since there is a separate code segment in the target program that handles each section of the PDF input, conducting additional mutations on sections with low weight, as facilitated by the weight strategy, contributes to exploring paths associated with the specific code segment.

Table 3 provides a detailed overview of the path coverage results of AFL and the Format-agnostic mutations (weight and resize strategies) on the libxml2 benchmark. The experiment was repeated five times, with each run lasting six hours. In the evaluation of AFL and Format-agnostic mutations for the libxml2 benchmark, which employs XML format inputs—a more intricate structure than PDF, the Uniform mutation method was not applied due to the complexity of XML syntax, including the presence of user-defined tags and attributes. On average, the Format-agnostic mutations (weight and resize strategies) achieved 101% and 105.4% higher path coverage, respectively, compared to AFL. These results highlight the effectiveness of the Format-agnostic mutation method in exploring path coverage by partitioning the input and applying varying numbers

of mutations to each section, even for inputs with complex formats like XML. Notably, the resize strategy outperformed the weight strategy in the case of XML. This can be attributed to the densely sized sections present in XML inputs, and the resizing strategy adeptly divides the input to accommodate these dense sections, thereby contributing to improved path coverage.

Table 3. The # of paths found by AFL and Format-agnostic mutation on libxml2.

Test Number	AFL	Format-agnostic (weight)	Format-agnostic (resize)
#1	1533	3447	3150
#2	1697	3249	3654
#3	2073	3686	3556
#4	1519	3219	3434
#5	1593	3314	3490
Avg	1683	3383	3456.8

Overall, the results demonstrate that the Uniform and Format-agnostic mutation methods offer significant advantages over AFL regarding path coverage. The Uniform mutation method, leveraging its prior knowledge of the input format structure, outperforms the other methods in achieving higher path coverage. However, the Format-agnostic mutation methods also exhibit notable path coverage by dynamically partitioning the input into sections based on coverage feedback, even without prior knowledge of the input format structure, regardless of its structure complexity. This highlights their effectiveness in adapting to different input scenarios and achieving satisfactory path coverage.

5 Related Work

Mutation-Based Fuzzing. Mutation-based fuzzing has been proposed to generate inputs by randomly modifying valid inputs. Some studies leverage heuristics to guide mutation. AFL [23], Angora [2], CollAFL [6], and Mopt [11] utilize coverage for the guidance, and MemFuzz [3] and MemLock [18] leverage memory access and memory usage. Mutation-based fuzzing shows high speed to generate inputs but, it is less effective for programs that use structured inputs. For example, fuzzers like Angora [2] and Qsym [22] rely on program context (e.g., branches), not the file context, there may still exist codes that are less tested for a given time.

Structured Input Fuzzing. Several approaches have been proposed to perform mutations based on grammar or specification to generate structured inputs. Squirrel [24], Superion [17], SD-Gen [14] leverage the AST based on input specifications and the grammar to generate the valid inputs. On the other hand, JANUS [20] and AFLTurbo [16] apply mutations on the metadata dimension intensively. They focus only on a specific segment of the structured input rather than the overall input segments.

6 Conclusion

In this paper, we have tackled the bias issue present in existing mutation methods utilized in coverage-based grey-box fuzzing. We introduced new mutation methods for structured input, namely Uniform and Format-agnostic. The Uniform mutation method addresses the bias by conducting additional mutations on the input sections that invoke less testing for the code segments in the target program responsible for handling those corresponding input sections. This method ensures more balanced path coverage across the target program. On the other hand, the Format-agnostic mutation method automatically divides the input into sections based on coverage feedback. Then it adjusts the number of mutations or section sizes according to the adopted strategy. Unlike the Uniform mutation method, the Format-agnostic mutation method does not rely on explicit format knowledge, making it more versatile for inputs with complex structures.

To evaluate the effectiveness of our proposed methods, we conducted experiments using two real-world programs (Xpdf and libxml2) and compared the results with AFL. The experimental outcomes demonstrated that our approaches surpassed AFL regarding path coverage exploration. The Uniform mutation method consistently achieved stable results with higher path coverage compared to AFL. Meanwhile, the Format-agnostic mutation method effectively partitioned the input into sections and successfully explored paths within target programs even when dealing with inputs featuring complex structural formats. Our proposed approaches effectively address the bias problem inherent in existing mutation methods, leading to improved path coverage while maintaining usability.

Acknowledgements. This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2023-2018-0-01441) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

References

1. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. **45**(5), 489–506 (2017)
2. Chen, P., Chen, H.: Angora: efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 711–725. IEEE (2018)
3. Coppik, N., Schwahn, O., Suri, N.: MemFuzz: using memory accesses to guide fuzzing. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 48–58. IEEE (2019)
4. Fan, R., Chang, Y.: Machine learning for black-box fuzzing of network protocols. In: Qing, S., Mitchell, C., Chen, L., Liu, D. (eds.) ICICS 2017. LNCS, vol. 10631, pp. 621–632. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89500-0_53
5. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: {AFL++}: combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 2020) (2020)

6. Gan, S., et al.: CollAFL: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 679–696. IEEE (2018)
7. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 474–484. IEEE (2009)
8. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: stateful black-box fuzzing of proprietary network protocols. In: Thuraisingham, B., Wang, X.F., Yegneswaran, V. (eds.) SecureComm 2015. LNICST, vol. 164, pp. 330–347. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28865-9_18
9. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 206–215 (2008)
10. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
11. Lyu, C., et al.: MOPT: optimized mutation scheduling for fuzzers. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1949–1966 (2019)
12. Peng, H., Shoshitaishvili, Y., Payer, M.: T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 697–710. IEEE (2018)
13. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: application-aware evolutionary fuzzing. In: NDSS, vol. 17, pp. 1–14 (2017)
14. Sargsyan, S., Kurmangaleev, S., Mehrabyan, M., Mishechkin, M., Ghukasyan, T., Asryan, S.: Grammar-based fuzzing. In: 2018 Ivannikov Memorial Workshop (IVMEM), pp. 32–35. IEEE (2018)
15. Serebryany, K.: {OSS-Fuzz}-Google's continuous fuzzing service for open source software (2017)
16. Sun, L., Li, X., Qu, H., Zhang, X.: AFLTurbo: speed up path discovery for greybox fuzzing. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pp. 81–91. IEEE (2020)
17. Wang, J., Chen, B., Wei, L., Liu, Y.: Superion: grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 724–735. IEEE (2019)
18. Wen, C., et al.: MemLock: memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 765–777 (2020)
19. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 511–522 (2013)
20. Xu, W., Moon, H., Kashyap, S., Tseng, P.N., Kim, T.: Fuzzing file systems via two-dimensional input space exploration. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 818–834. IEEE (2019)
21. Yue, T., et al.: {EcoFuzz}: adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In: 29th USENIX Security Symposium (USENIX Security 2020), pp. 2307–2324 (2020)
22. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: {QSYM}: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 745–761 (2018)
23. Zalewski, M.: American fuzzy lop (2020). https://lcamtuf.coredump.cx/afl/
24. Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., Wu, D.: SQUIRREL: testing database management systems with language validity and coverage feedback. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 955–970 (2020)