Visualizing and Understanding the Internals of Fuzzing

Sriteja Kummita
Fraunhofer IEM
Paderborn, Germany
sriteja.kummita@iem.fraunhofer.de

Zenong Zhang
University of Texas at Dallas
Richardson, TX, USA
zenong@utdallas.edu

Eric Bodden
Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

Shiyi Wei
University of Texas at Dallas
Richardson, TX, USA
swei@utdallas.edu

Abstract

Greybox fuzzing is one of the fuzzing techniques that has been extensively researched and used in practice. Plenty of publications propose improvements to greybox fuzzing. However, the extent to which these improvements really work and generalize is not yet understood: our preliminary study of the recent literature in greybox fuzzing shows that most papers evaluate their fuzzers in terms of runtime code coverage or bug-finding capability, although the improvements made are to the internal components (or internals) of the fuzzer. Results drawn from such experiments are insufficient to judge the impact the changes in the fuzzer's internals have on its performance.

To understand fuzzing better, we thus propose to evaluate fuzzers more in depth. To this extent, we suggest to develop (1) a fuzzing-specific visualization framework to support different analytic tasks that is scalable across multiple fuzzers and facilitates effective comparison of fuzzing internals, and (2) an evaluation specification to automate the evaluation process using visualization analysis.

Realizing this vision will allow us to finally answer the following questions: How can one effectively visualize and compare fuzzing internals? And what internal changes between the fuzzers are responsible for their performance deviations?

CCS Concepts

• **Software and its engineering** → *Software maintenance tools*; **Domain specific languages**; **Graphical user interface languages**.

Keywords

greybox fuzzing, evaluation, visualization analysis, domain-specific language

ACM Reference Format:
Sriteja Kummita, Zenong Zhang, Eric Bodden, and Shiyi Wei. 2024. Visualizing and Understanding the Internals of Fuzzing. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3691620.3695284

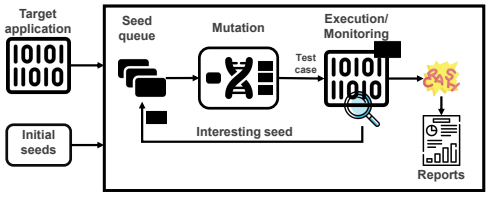


Figure 1: General process of greybox fuzzing.

1 Introduction

Fuzzing is a frequently used security-testing technique [2, 11, 51]. A fuzzer repeatedly generates inputs and executes the target application using these inputs, with the aim to find abnormal behaviors such as crashes or vulnerabilities. Greybox fuzzing (GF) uses runtime information, such as code coverage, to improve its input generation to increase runtime code coverage and find crashes or vulnerabilities. Figure 1 shows the general GF process.

A fuzzer[1] starts by executing the target application with the provided set of initial seeds.[2] During the execution, the target is monitored for interesting behaviour such as crashes or new runtime code coverage (Execution/Monitoring). After each execution, interesting seeds are stored in the seed queue. The search strategy algorithm selects a seed from the queue (Seed queue) and mutates it further with the hope that the mutants will trigger interesting behaviours (Mutation). This process continues until a stopping criterion such as a timeout is reached.

Numerous publications focus on improving GF in different areas [1, 4, 8, 12, 14, 16, 19, 22, 23, 25, 31, 32, 35, 37, 38, 40–45, 49, 50, 52]; however, none focus on explaining why or how the improvements work. Our preliminary study of 20 publications in GF shows that most claimed improvements are evaluated in terms of code coverage or bug-finding capability. Although it is tempting to infer that a fuzzer is efficient if it finds more bugs or executes more code, it is rather hard to understand how much any such results, when

[1] In the context of this paper, fuzzer refers to a tool that performs greybox fuzzing.
[2] A seed is an input that a fuzzer uses to execute the target application.

This work is licensed under a Creative Commons Attribution International 4.0 License.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695284

measured in terms of code coverage and bug detection, can be attributed to the investigated changes. A deeper understanding of the internal components (or internals) of fuzzing enables the development of better fuzzers and better evaluations.

To study the process in depth, some of the literature also focuses on visualizing different outcomes of a fuzzing campaign [7, 15, 48]. However, these visualizations only capture the high-level information such as covered code, the call graph, and interesting inputs generated over time. While this helps in visualizing the effects of fuzzing as a whole, it does not foster the understanding of the effects on the internals where improvements are made. Hence, *"how does fuzzing work?"* or specifically, *"what effects do alleged improvements to a fuzzer's internals have?"* still remain open questions.

We build up on this observation and formulate the main goals for our work as follows: (1) How can we effectively visualize and compare the internals of fuzzing? (2) What internal changes between the fuzzers are responsible for their performance deviations?

2 Motivation

2.1 Insufficient Evaluations

We conducted a preliminary study to capture the necessary information to (1) categorize greybox fuzzing into different (internal) stages with respect to the claims/improvements made in the publications and (2) map the claims/improvements and evaluation metrics to these extracted stages.

GF Internal Stage Categorization. We have carefully read different claims made in the literature and survey publications ([20, 26, 27, 30, 47]) in the area of greybox fuzzing to divide the fuzzing process into the internal stages as shown in the Figure 2: *instrumentation* modifies the target in order to get feedback from runtime [5, 21]; *initial seeds* are necessary to kick-start the fuzzing campaign and they influence the overall campaign [13, 21, 33]; *search strategy* decides which seed to select next from the seed queue using some heuristics [17]; *power schedule* assigns energy (number of mutations) to the selected seed using some heuristics (e.g., increased coverage) [47]; *mutations* apply different set of mutation strategies such as havoc, splice, etc., to the selected seed and produces a set of mutants until the assigned energy is exhausted [6, 26]; *execution* executes these mutants on the instrumented target, monitors for desired performance [30] and adds the interesting mutants (e.g., that provide increased coverage) to the seed queue.

Mapping information. Our preliminary study[3] consists of literature in the area of greybox fuzzing that provide improvements to at least one of the internal stages (Figure 2). We looked into each publication, especially, the *introduction* section where, the authors list their claims (contributions) and the *evaluation* section where the authors mention the metrics using which the claims are evaluated. For example, based on the following text in the *introduction* section of [31], *"We define innovative mutation operators that work on the...."* and *"We introduce a novel validity-based power schedule that enables..."*, we categorize the claims to *mutations* and *power schedule* internal stages; based on the following text in the *evaluation* section of the same publication ([31]), *"we investigate whether... exposes more unique crashes than..."* or *"whether ... explores more paths than ... in the given time budget."* or *"we investigate the number of bugs found by each technique..."*, we categorize the evaluation metrics to the *execution* internal stage. We followed the same approach for all the publications in our preliminary study to extract a map of claims and evaluation metrics to the internal stages.

[3]Our artifact [18] contains the list of 20 reviewed papers and the information on how the claims and the metrics are categorized and mapped to the internal stages.

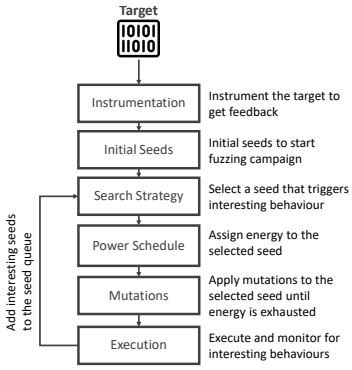


Figure 2: Internal stages of greybox fuzzing that influence the fuzzing performance.

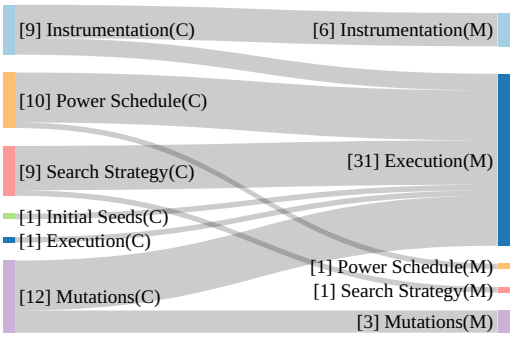


Figure 3: Relationship between the number of claims (C) and the frequency with which a given evaluation metric (M) was used with respect to internal fuzzing stages.

Using this mapping, we can discover the discrepancies between them. For example, one can easily find out if the claim and its corresponding evaluation metric(s) fall in the same internal stage or not. Figure 3 shows this information as the relationship between the number of claims (C) and evaluation metrics (M) with respect to the internal stages. This figure shows that most claims on improvements to fuzzing internal stages are evaluated with metrics that fall into the *execution* stage, such as code coverage, path coverage, time to exposure of a crash, etc., which do not correspond to the internal fuzzing data. A recent study of 250 fuzzing publications from the top conferenes also revealed that 77% of the approaches are evaluated using code coverage metrics and 71% evaluated using bug finding capability [34]. While we are not claiming that the current evaluations are incorrect, they are insufficient to fully understand the effect that the evaluated change to the fuzzer has within that fuzzer.

2.2 Visualizing fuzzing internals

Visualization helps in enhancing the human understanding of a phenomenon [29]. Consider an example of the visualization analysis workflow, shown in Figure 4, to visualize and compare different mutation phases in fuzzers. Figure 4a shows the runtime code coverage over time for two fuzzers, AFLFast [3] and AFLVanilla (a variant of AFLFast after replacing AFLFast's search strategy with the one from AFL 1.94b [46]), on the target program *cxxfilt*.[4] The fuzzing campaign is run for 24 hours and is repeated five times (one line for each run is shown for both fuzzers).

One can then visualize and compare only one run of the fuzzer (first run is highlighted with black solid line in the graph). We can see that AFLFast has slightly better coverage than AFLVanilla. The coverage growth is also steadily increasing between the 2nd hour and 7th hour (marked with an ellipse in the graph), which can be selected by the user to look into the underlying data.

Different mutants generated in the mutation phases are responsible for the coverage growth over time. Figure 4b captures exactly this information for the selected area in Figure 4a. It shows the distribution of coverage gain from different mutants generated in the two mutation phases, havoc and splice [6], for the two fuzzers. From the left half of the Figure 4b, it is not very evident which fuzzer has better coverage gain in the havoc mutation phase, when we consider the median values (marked with blue circle). However, when we look at the corresponding utilization ratio (ratio of the number of mutations that found the latest new coverage to the total number of mutations) shown in Figure 4c, it is clear that in the havoc mutation phase AFLFast shows a favourable performance. For the splice mutation phase, both Figures 4b (right half) and 4c show that AFLVanilla performs better. In Figure 4b we can also see that only few mutants have non-zero coverage gain, especially in the havoc mutation phase. To analyze deeper, one can visualize these mutants and their lineages.

But such a deeper analysis of fuzzing internals is impossible without proper visualizations. Furthermore, it is tedious and relatively hard to analyze the plots for each run and identify patterns across multiple runs. This is why we need a framework that can effectively visualize and summarize the information from multiple fuzzing runs and facilitate different analytic tasks. The visualizations should also be linked together and should coordinate with each other based on the user interactions. For example, the user should be able to select different areas in the plots to enable dynamic visualization and analysis. We believe such a framework can support both fuzzer users and developers: it helps fuzzer users not only to select a better fuzzer for their use case but also provides detailed insights on its internal workings; fuzzer developers can use such analyses for introspection and evaluation purposes.

No current visualization framework provides an infrastructure to visualize the internals of fuzzing and facilitate comparison between different fuzzers. The plots mentioned above are drawn using *ggplot2* [36], which provides an infrastructure to create static visualizations and is domain-agnostic. The visualizations in Figure 4 are only for demonstration purposes and should not be treated as the final design. We plan to develop a visualization framework for fuzzing by following the methodology detailed in Section 3.

[4] https://sourceware.org/binutils/docs/binutils/c_002b_002bfilt.html

3 Proposed Methodology and Evaluation

To systematically (re-)evaluate and understand fuzzing better, we propose to develop an evaluation framework for fuzzing. Figure 5 shows the overview of the proposed framework. The framework accepts a target and claims as input and generates different plots that support visualization and comparison analysis of the internal fuzzing data. The proposed approach has three main steps (contributions), as follows:

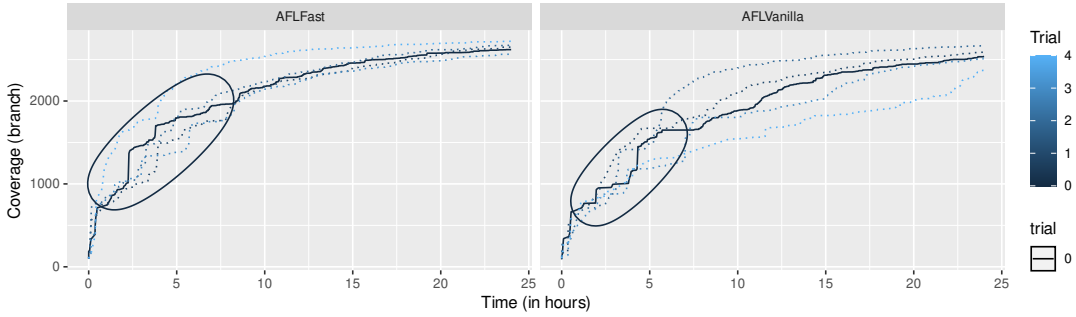
1. Monitor and record internal fuzzing data. Improvements to the overall performance of GF are generally made in one or more internal stages mentioned in Figure 2 by improving how the components handle the internal fuzzing data. For example, the power schedule calculates the number of mutations assigned to each selected seed [3]. Improvements in the power schedule often aim at assigning a larger number of mutations (energy) to the seeds that are more promising to generate new interesting behaviours. We can access the internal fuzzing data by instrumenting the source code of the fuzzer and take advantage of the fuzzer's logging module to export the internal data into a log file. For example, to record the power schedule, we can instrument the fuzzer's source code where the mutation takes place and insert code to log the number of mutations. However, manually instrumenting each internal stage in each fuzzer is a tedious process and does not scale well across multiple fuzzers.

To record data from multiple internal stages, there is also a need to define what data to be collected and tackle the scalability problem across multiple fuzzers. Hence, we propose to implement APIs that serve as an interface between the fuzzers and our evaluation framework. Such an interface helps in automating the process of internal data collection and scales across multiple fuzzers.

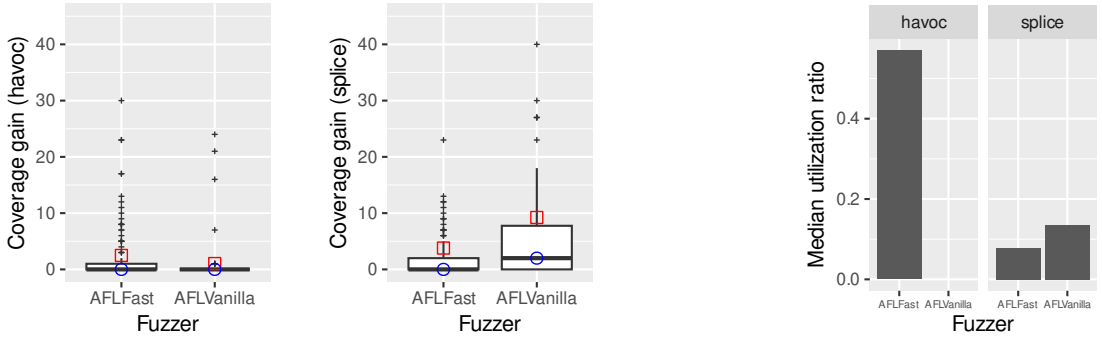
2. Visualizing the internals of fuzzing. Now that we have the infrastructure to capture the internals of fuzzing, we can generate different visualizations and perform many analytic tasks to understand the relationships between different internal fuzzing stages and also facilitate comparison between fuzzers.

Current visualization tools for fuzzing only capture high-level information such as reachable code, call graph, interesting test cases over time. VisFuzz provides a user interface to view the call graph and control-flow graph reachability in the browser and provides real-time intervention of the fuzzing process [48]. FMViz provides visualization to understand the mutation in AFL by exporting byte level changes to the input to an image [15]. FuzzSplore also provides visualizations about the mutations along with code coverage and interesting test cases generated over time [7]. However, none of the frameworks perform any data analysis on internal fuzzing data and facilitate comparison. To reduce randomness and obtain statistically significant results [17], fuzzing experiments must also be repeated multiple times. It is then tedious to generate plots for each run and manually analyze the data for patterns and relationships.

Therefore, we propose to develop a visualization framework that is specific to fuzzing. We plan to do this in three steps. First, investigate on how to compare internal fuzzing metrics using visualizations and how to scale the visualizations for multi-run fuzzing data, using aggregations, clustering, etc. We will start investigating according to the frameworks for visual comparison detailed by Gleicher et al in [9, 10]. These frameworks provide high-level



(a) Runtime branch coverage over time of 5 runs of the fuzzers, AFLFast and AFLVanilla on the target program *cxxfilt*. Each line in the graph represents a run and only first run is highlighted (solid line) in the graphs.



(b) Coverage gain of havoc and splice mutation phases in AFLFast and AFLVanilla for the selected Figure a areas (marked with an ellipse).

(c) Utilization ratio of havoc and splice phases in AFLFast and AFLVanilla for the selected Figure a areas.

Figure 4: Example visualizations of the internal mutation stages of AFLFast and AFLVanilla.

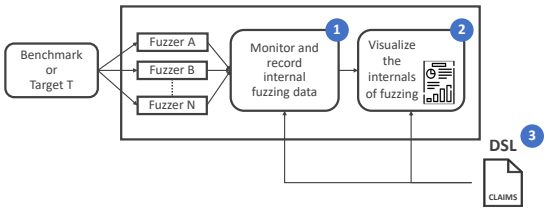


Figure 5: Overview of the proposed framework for fuzzer evaluation (the contributions of this work are enumerated).

design patterns for visual comparison. Second, develop a visualization framework to support the dynamic construction of linked and interactive visualizations of fuzzing metrics and mechanisms, and facilitate visual comparison methods for internal fuzzing data. We will follow the task-based approach by using the well-known nested model for visualization design by Munzner [28] to develop the visualization framework. As the first step, we will start by conducting interviews with fuzzing experts to understand their needs and extract a taxonomy of visualization analysis tasks that help in understanding fuzzing internals. These tasks can then be used to derive the necessary visualization idioms (scatterplots, grouped line plots, time series, etc) and the interactions between different idioms (brushing, zooming, etc). Third, develop a domain-specific grammar-based visualization toolkit for fuzzing. The goal of the toolkit is to enable fuzzing researchers and practitioners a means to flexibly compose interactive visualizations that answer their questions. The toolkit will also support the tasks in the aforementioned task taxonomy, and consider common visual comparison guidelines [9, 10] in the design process. The efforts lie in the direction similar to Gosling [24], a grammar-based toolkit for scalable and interactive genomics data visualization. We are unaware of any visualization framework that supports interactive visualization and analysis of internal fuzzing data.

3. Evaluation specification. As a final step, we propose to automate the evaluation process by developing an evaluation specification using a domain specific language (DSL). The DSL's purpose is three-fold: (1) it combines claims and evaluation metrics with the internal stages of fuzzing and ensures that the claims are evaluated directly, (2) it accommodates the visualization grammar using which users can specify what and how to generate and compose visualizations for different internal fuzzing metrics, and (3) it makes evaluations more easily reproducible.

One can use the DSL to provide all the necessary options for visualization analysis: the internal fuzzing data to capture, linked graphs and their interactions (brushing, selection window, etc) to generate. We aim to develop a flexible DSL to accomodate user-defined evaluation metrics along with the existing ones in the

literature and custom interactive visualiaztions which support user-desired visualization analyses.

Research questions. We plan to evaluate our approach by answering these research questions:

(1) What visualizations/analyses help in understanding fuzzing better and how?
(2) What are the scalability challenges faced in comparing different fuzzing metrics?
(3) How usable is the visualization framework to understand internals of fuzzing?
(4) How expressible is the evaluation specification?

Threats to Validity. We currently foresee the following threats: the properties of the target influence the fuzzing performance [39]. Investigating deeper in this direction is out of scope for our research idea. However, we keep this investigation as an optional contribution.

Our proposed approach involves instrumenting the fuzzers to gather internal data which influence the runtime performance. To mitigate this, we plan to develop a configurable instrumentation methodology that captures only necessary and sufficient data for visualization analysis at intermittent intervals during the fuzzing campaign.

4 Conclusion

Greybox fuzzing (GF), in general, aims at increasing runtime code coverage and finding more bugs. There are many publications that improve different stages of GF (discussed in Section 2.1). However, detailed understanding and visualizing the internals of fuzzing still remains an open challenge. From the perspective of understanding fuzzing, we also identified the discrepancy between the claims and the evaluation metrics in our preliminary study (as shown in Figure 3). We then detailed our approach, research questions and contributions that focus on visualizing and understanding the internals of fuzzing. We believe that the contributions of this work will help in understanding fuzzing better and support the evaluation process in the fuzzing community.

Acknowledgments

This work was partly supported by the Fraunhofer Internal Programs under Grant No. PREPARE 840 231, and NSF grants CCF-2008905 and CCF-2047682.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *NDSS*, Vol. 19. 1–15. https://doi.org/10.14722/ndss.2019.23371
[2] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (May 2021), 79–86. https://doi.org/10.1109/MS.2020.3016773
[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[4] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2095–2108. https://doi.org/10.1145/3243734.3243849
[5] James Fell. 2017. A review of fuzzing tools and methods. *PenTest Magazine* (2017).
[6] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. 2023. Dissecting American Fuzzy Lop: A FuzzBench Evaluation. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 52 (mar 2023), 26 pages. https://doi.org/10.1145/3580596
[7] Andrea Fioraldi and Luigi Paolo Pileggi. 2021. FuzzSplore: Visualizing Feedback-Driven Fuzzing Techniques. *arXiv preprint arXiv:2102.02527* (2021).
[8] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040
[9] Michael Gleicher. 2018. Considerations for Visualizing Comparison. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 413–423. https://doi.org/10.1109/TVCG.2017.2744199
[10] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D Hansen, and Jonathan C Roberts. 2011. Visual comparison for information visualization. *Information Visualization* 10, 4 (2011), 289–309. https://doi.org/10.1177/1473871611416549
[11] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.
[12] Yubo He and Yuefei Zhu. 2023. RLTG: Multi-targets directed greybox fuzzing. *PLOS ONE* 18, 4 (2023), 1–23. https://doi.org/10.1371/journal.pone.0278138
[13] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795
[14] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. 36–50. https://doi.org/10.1109/SP46214.2022.9833751
[15] Aftab Hussain and Mohammad Amin Alipour. 2021. FMViz: Visualizing Tests Generated by AFL at the Byte-level. arXiv:2112.13207 [cs.SE]
[16] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4931–4948.
[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804
[18] Sriteja Kummita, Zenong Zhang, Eric Bodden, and Shiyi Wei. 2024. Preliminary Study. https://zenodo.org/records/13710474
[19] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. 2023. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 384–396. https://doi.org/10.1109/ICSE48619.2023.00043
[20] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (Dec. 2018), 6. https://doi.org/10.1186/s42400-018-0002-y
[21] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (Sept. 2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476
[22] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. 2019. Sequence Coverage Directed Greybox Fuzzing. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 249–259. https://doi.org/10.1109/ICPC.2019.00044
[23] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2021. DeepFuzzer: Accelerated Deep Greybox Fuzzing. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2021), 2675–2688. https://doi.org/10.1109/TDSC.2019.2961339
[24] Sehi LYi, Qianwen Wang, Fritz Lekschas, and Nils Gehlenborg. 2021. Gosling: A grammar-based toolkit for scalable and interactive genomics data visualization. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2021), 140–150. https://doi.org/10.1109/TVCG.2021.3114876
[25] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1024–1036. https://doi.org/10.1145/3377811.3380421
[26] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563
[27] Lucas McDonald, Muhammad Ijaz Ul Haq, and Ashley Barkworth. 2021. Survey of Software Fuzzing Techniques. (2021).

[28] Tamara Munzner. 2009. A Nested Model for Visualization Design and Validation. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (Nov. 2009), 921–928. https://doi.org/10.1109/TVCG.2009.111

[29] Tamara Munzner. 2014. *Visualization analysis and design.* CRC press.

[30] Mathias Payer. 2019. The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes. *IEEE Security & Privacy* 17, 1 (Jan. 2019), 78–82. https://doi.org/10.1109/MSEC.2018.2889892

[31] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

[32] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, and Zhenyu Chen. 2023. DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop* (Seattle, WA, USA) *(FUZZING 2023)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3605157.3605172

[33] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 861–875.

[34] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1974–1993. https://doi.org/10.1109/SP54263.2024.00137

[35] Lei Sun, Xumei Li, Haipeng Qu, and Xiaoshuai Zhang. 2020. AFLTurbo: Speed up Path Discovery for Greybox Fuzzing. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 81–91. https://doi.org/10.1109/ISSRE5003.2020.00017

[36] Randle Aaron M. Villanueva and Zhuo Job Chen. 2019. *ggplot2: Elegant Graphics for Data Analysis (2nd ed.)*. Vol. 17. Routledge. 160–167 pages. https://doi.org/10.1080/15366367.2019.1565254

[37] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 999–1010. https://doi.org/10.1145/3377811.3380386

[38] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

[39] Dylan Wolff, Marcel Böhme, and Abhik Roychoudhury. 2022. Explainable Fuzzer Evaluation. (2022). arXiv:2212.09519 [cs.SE]

[40] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. Enhancing Coverage-Guided Fuzzing via Phantom Program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1037–1049. https://doi.org/10.1145/3611643.3616294

[41] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 56–68. https://doi.org/10.1109/ICSE48619.2023.00017

[42] Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. 2024. Fine-grained Coverage-based Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 138 (jun 2024), 41 pages. https://doi.org/10.1145/3587158

[43] Xiaofei Xie, Hongxu Chen, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Coverage-Guided Fuzzing for Feedforward Neural Networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1162–1165. https://doi.org/10.1109/ASE.2019.00127

[44] Tai Yue, Yong Tang, Bo Yu, Pengfei Wang, and Enze Wang. 2019. Learnafl: Greybox fuzzing with knowledge enhancement. *IEEE Access* 7 (2019), 117029–117043. https://doi.org/10.1109/ACCESS.2019.2936235

[45] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324.

[46] Michal Zalewski. 2013. American fuzzy lop. https://github.com/mirrorer/afl

[47] Xiaoqi Zhao, Haipeng Qu, Jianliang Xu, Xiaohui Li, Wenjie Lv, and Gai-Ge Wang. 2024. A systematic review of fuzzing. *Soft Computing* 28, 6 (2024), 5493–5522. https://doi.org/10.1007/s00500-023-09306-2

[48] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, Chengnian Sun, and Yu Jiang. 2019. VisFuzz: Understanding and Intervening Fuzzing with Interactive Visualization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1078–1081. https://doi.org/10.1109/ASE.2019.00106

[49] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. https://doi.org/10.1145/3460120.3484596

[50] Xiaogang Zhu, Xiaotao Feng, Xiaozhu Meng, Sheng Wen, Seyit Camtepe, Yang Xiang, and Kui Ren. 2022. CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2022), 912–923. https://doi.org/10.1109/TDSC.2020.3008826

[51] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345

[52] Yanyan Zou, Wei Zou, JiaCheng Zhao, Nanyu Zhong, Yu Zhang, Ji Shi, and Wei Huo. 2023. PosFuzz: augmenting greybox fuzzing with effective position distribution. *Cybersecurity* 6, 1 (June 2023), 11. https://doi.org/10.1186/s42400-023-00143-2