2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)

SEEDUI: Understanding Initial Seeds in Fuzzing

Sriteja Kummita*, Eric Bodden*, Miao Miao†, Shiyi Wei†
* *Heinz Nixdorf Institute, Paderborn University, Fraunhofer IEM, Paderborn, Germany*
{sriteja.kummita, eric.bodden}@uni-paderborn.de
† *University of Texas at Dallas, Richardson, TX, USA*
{mmiao, swei}@utdallas.edu

***Abstract*—Greybox fuzzing, a widely used dynamic testing technique, iteratively tests the software using semi-randomly generated inputs that aim at reaching more code at runtime. It relies on a set of initial inputs (initial seeds) to bootstrap the fuzzing process. These initial seeds highly influence the fuzzing performance in terms of runtime coverage or finding bugs.**

This paper introduces SEEDUI, a tool that helps in visualizing and understanding the performance of initial seeds across multiple fuzzing campaigns. It provides five different views that help a user for an in-depth initial seed analysis.

In our previous work, we extracted a visualization task taxonomy for greybox fuzzing by interviewing 33 fuzzing experts and analyzing the responses. We evaluate SEEDUI along with four existing tools: VisFuzz, FuzzSplore, FuzzInspector, and Ijon UI, in terms of their support to the taxonomy. Our findings indicate that SEEDUI complements existing tools by addressing 9 tasks in the taxonomy that are not supported by them.

Demonstration video: https://youtu.be/qpPjutmIcTs
Artifacts: https://github.com/secure-software-engineering/SeedUI

***Index Terms*—visualization, greybox fuzzing, initial seeds**

I. INTRODUCTION

Greybox fuzzing (GF), a dynamic testing technique, aims at increased runtime coverage and bug-finding capability in a system under test (SUT). A greybox fuzzer (e.g., AFL++ [1]) uses a set of valid initial inputs (initial seeds) for SUT to kick-start the fuzzing process. Starting with the initial seeds, it iteratively derives new inputs (seeds) that aim at increased runtime coverage or finding bugs. The new inputs are derived either by applying random mutations on a single seed or by combining multiple seeds (using splicing[1]) or both.

The quality and the number of initial seeds highly influence the fuzzing performance [2]: the more diverse the initial seeds are (in terms of runtime coverage), the better the fuzzing yield [3]. Many approaches have been proposed to identify initial seeds or to minimize seeds from a previous fuzzing campaign [3], [4]. Tools such as FuzzSplore [5] and VisFuzz [6] help fuzzing practitioners visually understand the fuzzing process.

In this space, we introduce SEEDUI, a visualization tool that helps in deeper analysis of initial seed(s) performance across multiple fuzzing campaigns (runs) on a single SUT. SEEDUI differs from existing visualization tools in terms of its design and the supported usage scenarios. It provides five interactive views that can help a user analyze the initial seeds in terms of runtime coverage, the number and quality of its derived seeds, and the most mutated bytes of the initial seeds by the fuzzer. It can also help fuzzing practitioners or users visually understand the evolution of initial seeds overtime in a fuzzing campaign. The current implementation of SEEDUI supports all the fuzzers that are based on AFL++ [1].

[1]Splicing, as it is implemented in AFL++: https://tinyurl.com/23bpc7st

We evaluated SEEDUI with the existing fuzzing visualization tools in terms of their support towards the taxonomy of visualization analysis tasks for GF [7]. Our findings reveal that SEEDUI complements the existing tools by supporting in-depth analysis of initial seeds and can support 9 different tasks which the existing tools do not.

The rest of the paper is organized as follows: Section II provides detailed overview of SEEDUI followed by evaluation in Section III, limitations and future work in Section IV, and the conclusion in Section V.

II. SEEDUI

The purpose of SEEDUI is to help novice fuzzing users in visually understanding the performance of initial seeds across multiple fuzzing runs on a single SUT.

Figure 1 shows the overview of SEEDUI, which comprises of three modules: `preprocessing`, `data analyzer`, and `web interface`. A user can provide the corpus[2] from one or more AFL++ fuzzing runs along with the SUT compiled with debug information[3] to the SEEDUI, and use the `web interface` to gather insights on initial seeds. We next explain each of the three components in detail.

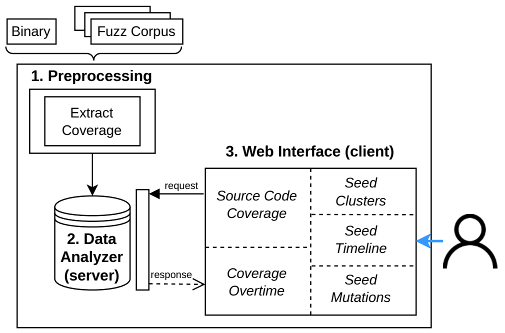


Fig. 1: High-level architecture and workflow of SEEDUI.

[2]A corpus refers to all the new inputs (seeds) saved in a fuzzing run.
[3]For example, using *-g* flag for clang and gcc compilers.

A. *Preprocessing*

The `preprocessing` module involves extracting the line coverage for the gathered corpora. SEEDUI relies on the dynamic binary instrumentation client, `drcov`, to extract line coverage for two reasons: `drcov` is compiler independent and flexible [8] as compared to the generally used coverage tools like gcov and llvm-cov, and AFL++ provides off-the-shelf support to generate `drcov` coverage trace information.[4]

Along with the extracted line coverage, information about the SUT, fuzzing corpora are then provided as inputs to `data analyzer`.

B. *Data Analyzer*

The `data analyzer` acts as a server and performs the following functions:

a) Coverage parsing. It parses the `drcov` coverage information and maps it to the corresponding lines in the source code. It also extracts the edge coverage recorded by the fuzzer.

b) Family tree extraction. For each initial seed in the corpus, `data analyzer` identifies all of its derived seeds and extracts a family tree. For convenience, we use the term, descendant, to refer to a derived seed.

c) Mutations computation. `data analyzer` tracks how many times each initial seed byte is modified across all its descendants.

`data analyzer` expects the corpora to include the following information: a unique identifier for each seed, UNIX time in milliseconds at which the seed is generated, identifiers of the seed precedents, and the coverage recorded by the fuzzer when it is mutated.

AFL++ provides all these identifiers except the seed generation time. We included a Git patch[5] in our artifact that can be used for AFL++-based fuzzers.

C. *Interactive Web Interface*

The client module, `web interface`, contains five independent views that constantly communicate with `data analyzer` based on the user interactions. The data between the views can be correlated using the unique seed identifiers.

Figure 2 shows an example of the five views: *source code coverage* (A), *fuzzer coverage overtime* (B), *seed clusters* (C), *seed timeline* (D), and *byte-wise seed mutations* (E). The data in this example was extracted from five 24-hour fuzzing runs on *readelf* program of *GNU binutils*[6], each with five different initial seeds.

1) Source code coverage: This view summarizes the coverage statistics of the SUT across the fuzzing runs (A1 in Figure 2). The user can click on an SUT file to view the corresponding line coverage. For deeper analysis of initial seeds, the user can select the initial seed and their corresponding descendant using the two dropdowns (A2 in Figure 2). These dropdowns facilitate the comparison of line coverage between the selected seeds. The coverage information is also color coded to ease the visual comparison (in A3 in Figure 2, blue highlights indicate the coverage from the initial seed, orange from its selected descendant, and green from both). This view allows the user to identify seeds for the subsequent fuzzing runs that reach the desired areas in the source code.

2) Fuzzer coverage overtime: This view shows the edge coverage recorded by the fuzzers overtime (B in Figure 2). Note that the edge coverage is different from the line coverage shown in the `source code coverage`. AFL++ assigns an ID to each basic block in the SUT and records the transitions (edge coverage) between them in a bitmap shared between the fuzzer and the SUT [9]. To record a transition between two basic blocks, a hash, computed using their IDs, is used to index into the shared bitmap to increment the corresponding bit. After executing each seed, AFL++ compares the shared bitmap with a global bitmap to identify new edge coverage.

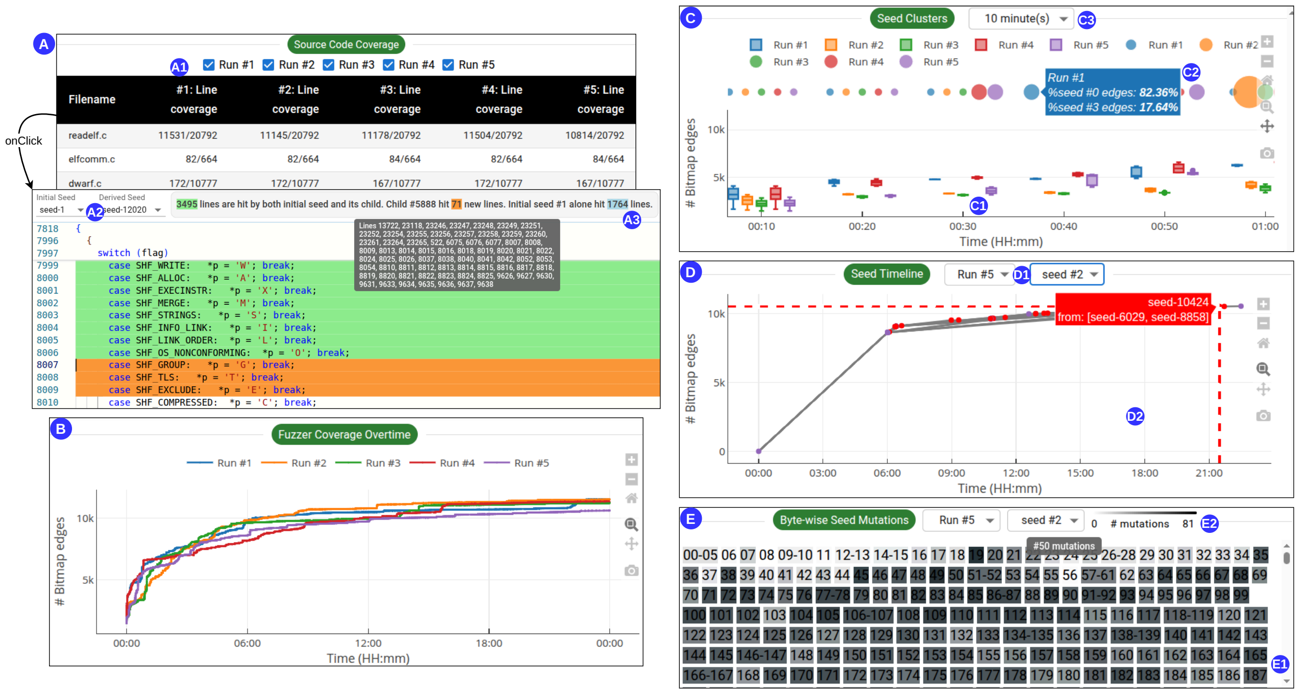
This view is necessary to identify the coverage plateaus across fuzzing runs, i.e., the time in which a fuzzer could not reach new areas in the SUT. Existing tools such as FuzzSplore [5], VisFuzz [6], and FuzzBench [10] also provide this view.

3) Seed clusters: This view summarizes the distribution of edge coverage recorded by the fuzzer(s) across time-clusters using box plots. As an example, Figure 2 C1 shows a series of box plots grouped according to the 10-minute clusters. Each box plot shows the distribution of edge coverage from the previous 10 minutes in the corresponding fuzzing run (e.g., the five box plots at 00:20 summarizes the edge coverage between 00:10 and 00:20 in the corresponding five fuzzing runs). For each box plot, a corresponding bubble is shown on top, representing the contribution of initial seeds in the fuzzing run in that time-cluster.

The contribution of each initial seed in a cluster is computed by taking the ratio of the total coverage of its descendants in the cluster to the total coverage of all the seeds in the cluster. The bubble size denotes the number of initial seeds involved in the corresponding time-cluster: the larger bubbles correspond to a greater number of initial seeds used by the fuzzer in the cluster. For example, at 30-minute cluster (00:30 in Figure 2 C1), two initial seeds contributed to the corresponding edge coverage distribution in Run #4 and Run #5 whereas in the other runs there was only one initial seed. On hovering over the bubble, a note is displayed with the initial seed contributions (C2 in Figure 2). The user can interactively analyze the data for different clusters sizes using the dropdown (C3 in Figure 2).

4) Seed timeline: This view displays the initial seed(s) evolution timeline in a fuzzing run (D in Figure 2). Once an initial seed of a fuzzing run is selected using the two dropdowns (D1 in Figure 2), a line graph is shown representing its evolution timeline (D2 in Figure 2). The lines between points represent the parent-child relationship whereas the position of each point gives the seed generation time and its edge coverage. A red point denotes a descendant derived from more than one seed. On hovering over the point, a note is displayed with the parent(s) information. The dropdown also allows the user to

[4]https://tinyurl.com/2s43nz95
[5]Patch is extracted from the AFL++ release with commit hash `b89727b`.
[6]https://sourceware.org/binutils/



Run #1 ✓ Run #2 ✓ Run #3 ✓ Run #4 ✓ Run #5 ✓

Filename	#1: Line coverage	#2: Line coverage	#3: Line coverage	#4: Line coverage	#5: Line coverage
readelf.c	11531/20792	11145/20792	11178/20792	11504/20792	10814/20792
elfcomm.c	82/664	82/664	84/664	82/664	84/664
dwarf.c	172/10777	172/10777	167/10777	172/10777	167/10777

Initial Seed: seed-1 Derived Seed: seed-12020

3495 lines are hit by both initial seed and its child. Child #5888 hit 71 new lines. Initial seed #1 alone hit 1764 lines.

Lines 13722, 23118, 23246, 23247, 23248, 23249, 23251, 23252, 23254, 23255, 23256, 23257, 23258, 23259, 23260, 23261, 23264, 23265, 522, 6075, 6076, 6077, 8007, 8008, 8009, 8013, 8014, 8015, 8016, 8018, 8019, 8020, 8021, 8022, 8024, 8025, 8026, 8037, 8038, 8040, 8041, 8042, 8052, 8053, 8054, 8810, 8811, 8812, 8813, 8814, 8815, 8816, 8817, 8818, 8819, 8820, 8821, 8822, 8823, 8824, 8825, 9626, 9627, 9630, 9631, 9633, 9634, 9635, 9636, 9637, 9638

```
7818  {
7996    {
7997      switch (flag)
7999        case SHF_WRITE:   *p = 'W'; break;
8000        case SHF_ALLOC:   *p = 'A'; break;
8001        case SHF_EXECINSTR:   *p = 'X'; break;
8002        case SHF_MERGE:   *p = 'M'; break;
8003        case SHF_STRINGS:   *p = 'S'; break;
8004        case SHF_INFO_LINK:   *p = 'I'; break;
8005        case SHF_LINK_ORDER:  *p = 'L'; break;
8006        case SHF_OS_NONCONFORMING:  *p = 'O'; break;
8007        case SHF_GROUP:   *p = 'G'; break;
8008        case SHF_TLS:   *p = 'T'; break;
8009        case SHF_EXCLUDE:   *p = 'E'; break;
8010        case SHF_COMPRESSED:  *p = 'C'; break;
```

Fig. 2: Five views in `web interface` module of SEEDUI.

select multiple initial seeds to visually understand the evolution timeline of the seeds derived from the selected combination.

5) Seed mutations: Using this view, users can inspect the byte-wise modifications that the fuzzer performed on the initial seed (E in Figure 2). On selecting an initial seed using the dropdowns, a sequence of highlighted numbers representing the bytes of the initial seed are displayed (E1 in Figure 2). The intensity of the background color for each byte represents the number of mutations performed by the fuzzer: the darker the color, the more the mutations. A gradient scale represents the mutations range for the selected initial seed (E2 in Figure 2). Similar to the other views, hovering on the byte(s) shows a note with the corresponding number of mutations.

III. EVALUATION

We evaluated SEEDUI with four existing visualization tools:

- VisFuzz helps in identification of source code bottlenecks that hinder the fuzzer's performance using callgraph and control-flow graph visualizations [6].
- FuzzSplore [5] helps the user in selecting a best fuzzing configuration for the SUT using four plots.
- FuzzInspector [11] provides an interface to monitor the fuzzing progress and memory contents of the CPU registers. It also allows the user to specify input generation constraints during the fuzzing campaign.
- Ijon UI [12] helps in monitoring the fuzzing progress and navigating through the source code covered by the fuzzer.

In our previous work, we interviewed 33 fuzzing experts for their desired visualization analysis tasks to understand fuzzing [7]. Based on their responses, we extracted a taxonomy of visualization analysis tasks, which can be used to evaluate fuzzing visualization tools. A tool is considered to support a task if it provides the necessary interfaces to visualize and interpret the relevant data to perform the task. For the evaluation, we consider all the analysis tasks that are supported by at least one of the visualization tools. Table I shows the 16 analysis tasks along with the corresponding tool support.

The first column describes the task and the rest of the columns specify the tool support information. For SEEDUI, we specify the corresponding view(s) that support the task and for existing tools we use the icon, ✓, to indicate their support for the task. SEEDUI is capable of supporting 12 tasks, of which existing tools can support only 3 tasks. However, there are 4 tasks that SEEDUI does not support, yet existing tools do. Hence, SEEDUI complements the existing fuzzing visualization tools by additionally providing interfaces to understand the initial seed(s) performance in the fuzzing campaigns.

Discussion. FuzzSplore also visualizes seed evolution using a generations graph (Task 5) and only provides information about their generated time. Additionally showing their corresponding coverage and enabling analysis of different initial seed combinations like SEEDUI does (Section II-C4) can help user in performing a deeper analysis of initial seeds.

All the tools except FuzzSplore support the comparison of coverage across all seeds (Task 90), however, they use different representations. SEEDUI and Ijon UI use coverage-annotated source code. Additionally, SEEDUI provides comparison capabilities between initial seeds and their descendants

TABLE I: Comparison of fuzzing visualization tools using a subset of our taxonomy [7].

Task Description (The number before each task description refers to Task ID in our taxonomy)	**SEEDUI**	**VisFuzz**	**Fuzz-Splore**	**Fuzz-Inspector**	**Ijon UI**
3. Comparison of coverage between different initial seed sets and their corresponding mutants from multiple fuzzing runs	Source code coverage, Seed clusters	-	-	-	-
4. Comparison of coverage among initial seeds in a single fuzzing run	Source code coverage	-	-	-	-
27. Comparison between the mutated parts among initial seeds	Seed mutations	-	-	-	-
30. Comparison of coverage between different initial seed sets from multiple fuzzing runs	Source code coverage	-	-	-	-
33. Visualize the source code regions that are not covered by the initial seeds at runtime	Source code coverage	-	-	-	-
34. Distribution of code coverage across different initial seed sets from multiple fuzzing runs	Seed clusters	-	-	-	-
81. Visualize the mutated parts of the seeds from a single fuzzing run	Seed mutations	-	-	-	-
66. Relationship between each mutation familiy and their coverage	Seed timeline	-	-	-	-
59. Comparison of coverage between a mutant and offsprings	Source code coverage, Seed timeline	-	-	-	-
5. Visualize the mutation timeline of the initial seeds along with their corresponding coverage	Seed timeline	-	✓	-	-
36. Relationship between initial seed sets and seed corpus size	-	-	✓	-	-
58. Comparison of the runtime similarity of the seeds and the similarity in terms of the reached source code regions	-	-	✓	✓	-
90. Comparison of coverage information across all seeds	Source code coverage	✓	-	✓	✓
91. Visualize the number and duration of executions of all seeds	-	✓	-	-	✓
92. Visualize the coverage bottlenecks in the source code	-	✓	-	✓	-
96. Visualize coverage of seeds overtime in a fuzzing run	Fuzzer coverage over-time	✓	✓	-	-

(Section II-C1). VisFuzz uses source code juxtaposed with coverage-annotated callgraphs and control-flow graphs. FuzzInspector uses block coverage and coverage-annotated flowchart of disassembled addresses to represent control-flow graphs.

The scope of this evaluation was limited to the tools that have a user interface and help understand fuzzing. As such, we avoid direct comparisons with the following tools that do not satisfy these requirements: FMViz [13] only helps in understanding the mutation process of AFL by exporting sequence of seeds to images. Fuzz-Introspector[7] focuses on identifying the fuzzing bottlenecks than visually understanding fuzzing. And FuzzBench [10] focuses on evaluating and ranking fuzzers based on their performance on real-world projects.

IV. Limitations and Future Work

We identified the following three limitations of SEEDUI: First, the usability of SEEDUI is not evaluated in this work. We plan to do such an evaluation in the future. However, evaluation in terms of our taxonomy shows that SEEDUI supports tasks that existing tools do not.

Second, SEEDUI does not support all the tasks that involve initial seeds in the taxonomy. To support these remaining tasks, fuzzers should be instrumented to extract the data about the fuzzing mechanisms such as power schedule [14], [15], and search strategy [16], [17]. In the future, we plan to extract the necessary data and extend SEEDUI to support these remaining tasks.

[7]https://github.com/ossf/fuzz-introspector

Third, SEEDUI works for all AFL++-based fuzzers. Currently, all the evaluated visualization tools including SEEDUI are based only on AFL++-based fuzzers. We aim to extend SEEDUI to include other fuzzers in the future.

V. Conclusion

In this paper, we introduced SEEDUI, a visualization tool that can help novice fuzzing practitioners or users to understand initial seed(s) performance across multiple AFL++ fuzzing campaigns on a single SUT. It provides five interactive views that can help the user to perform deeper analysis and gather insights into the seeds.

When compared to the existing fuzzing visualization tools using a subset of our visualization taxonomy for greybox fuzzing, SEEDUI supports 9 tasks which are not supported by any other existing tools. Hence, SEEDUI can complement these tools by additionally providing information about the initial seeds.

Acknowledgment

This work was partly supported by NSF grant CCF-2047682.

References

[1] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[2] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1974–1993.

[3] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 230–243.

[4] A. Herrera, H. Gunadi, L. Hayes, S. Magrath, F. Friedlander, M. Sebastian, M. Norrish, and A. L. Hosking, "Corpus distillation for effective fuzzing: A comparative evaluation," *arXiv preprint arXiv:1905.13055*, 2019.

[5] A. Fioraldi and L. P. Pileggi, "FuzzSplore: Visualizing feedback-driven fuzzing techniques," 2021.

[6] C. Zhou, M. Wang, J. Liang, Z. Liu, C. Sun, and Y. Jiang, "VisFuzz: Understanding and intervening fuzzing with interactive visualization," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1078–1081.

[7] S. Kummita, M. Miao, E. Bodden, and S. Wei, "Visualization task taxonomy to understand the fuzzing internals," *ACM Transactions on Software Engineering and Methodology*, 2025.

[8] M. A. Ben Khadra, D. Stoffel, and W. Kunz, "Efficient binary-level coverage analysis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1153–1164.

[9] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[10] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "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, p. 1393–1403.

[11] H.-L. Lu, R.-J. Zhuang, and S.-K. Huang, "Fuzz testing process visualization." *Journal of Information Science & Engineering*, vol. 39, no. 5, 2023.

[12] C. Aschermann and S. Schumilo, "Ijon UI." [Online]. Available: https://github.com/eqv/fuzz_ui

[13] A. Hussain and M. A. Alipour, "FMViz: Visualizing tests generated by afl at the byte-level," 2021.

[14] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 678–689.

[15] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[16] H. L. Nguyen and L. Grunske, "Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 249–261.

[17] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.