FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer

Stephan Lipp
Technical University of Munich
Germany

Daniel Elsner
Technical University of Munich
Germany

Thomas Hutzelmann
Technical University of Munich
Germany

Sebastian Banescu
Technical University of Munich
Germany

Alexander Pretschner
Technical University of Munich
Germany

Marcel Böhme
MPI-SP, Germany
Monash University, Australia

ABSTRACT

Performing sound and fair fuzzer evaluations can be challenging, not only because of the randomness involved in fuzzing, but also due to the large number of fuzz tests generated. Existing evaluations use code coverage as a proxy measure for fuzzing effectiveness. Yet, instead of considering coverage of all generated fuzz inputs, they only consider the inputs stored in the fuzzer queue. However, as we show in this paper, this approach can lead to biased assessments due to path collisions. Therefore, we developed FuzzTastic, a fuzzer-agnostic coverage analyzer that allows practitioners and researchers to perform uniform fuzzer evaluations that are not affected by such collisions. In addition, its time-stamped coverage-probing approach enables frequency-based coverage analysis to identify barely tested source code and to visualize fuzzing progress over time and across code. To foster further studies in this field, we make FuzzTastic, together with a benchmark dataset worth ~12 CPU-years of fuzzing, publicly available; the demo video can be found at `https://youtu.be/Lm-eBx0aePA`.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

fuzzing, benchmarking, software security

ACM Reference Format:
Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. 2022. FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. `https://doi.org/10.1145/3510454.3516847`

1 INTRODUCTION

Context. Detecting bugs, especially in programs written in error-prone languages such as C/C++, is an essential part of software testing. One dynamic testing approach that has enjoyed great popularity in recent years is known as fuzzing [16, 17]. The idea of fuzzing is simple: generate a large number of random inputs (mechanized by so-called fuzzers), feed them into a target program, and then see if it crashes, or not. Certain program crashes thereby enable attackers to execute malicious source code, leak sensitive data, or provoke a denial-of-service (DoS), and are therefore considered security vulnerabilities.

Problem and State-of-Practice. As highlighted by Klees *et al.* [11], conducting sound and fair fuzzer evaluations is difficult as they require long timeouts and numerous repetitions to account for the inherent randomness of fuzzing. Yet many of the existing evaluations do not meet these requirements, leading to incorrect or misleading assessments [11]. FuzzBench [15], a benchmarking service launched by Google, addresses these issues by conducting uniform and reproducible fuzzer evaluations using code coverage as proxy metric for fuzzing effectiveness. To measure code coverage, FuzzBench captures the inputs from the fuzzer queue[1] [11, 14] and then replays them on a second instance of the target program that is compiled with Clang's code coverage feature [2] enabled.

However, this approach has two limitations. Firstly, it inhibits measuring the exact number of executions of a code region (*e.g.*, basic block) by the fuzz inputs, which would allow to reveal less thoroughly tested code. Secondly, it assumes that the inputs stored in the fuzzer queue correctly reflect the code coverage achieved by all generated fuzz inputs. However, due to path collisions in coverage-based fuzzers that hash the tracked coverage information [8, 9, 20] (discussed in Section 4), not all coverage-increasing inputs are added to the queue. Consequently, when measuring coverage from the inputs in the queue, code that was actually covered by the generated fuzz inputs may then be missed, resulting in distorted evaluations.

Solution and Contributions. In this paper, we present our LLVM-based coverage analyzer, called FuzzTastic, which works independently of the chosen fuzzer (black-, grey-, or whitebox). Besides addressing the limitations described above, our tool also enables visualizing the progress of covered code throughout the fuzzing campaign. Furthermore, we demonstrate FuzzTastic's applicability by creating a large-scale benchmark dataset, containing detailed coverage data from 9 fuzzers executed on 12 different C/C++ open-source programs, which can be used in further studies. Accordingly, this work presents the following *contributions*:

(1) We develop the coverage analysis tool FuzzTastic, which we make publicly available as open-source software on GitHub[2] (see Section 2).

[1] A fuzzer queue holds fuzz inputs, which increased coverage, for future usage [11].
[2] `https://github.com/tum-i4/fuzztastic`

(2) We generate and release a comprehensive fuzzer benchmark dataset with detailed coverage data worth ~12 CPU-years[3] of fuzzing (see Section 3).
(3) We show that path collisions in AFL-based fuzzers can cause an error of up to 9% missed basic blocks when measuring code coverage, introducing a non-negligible bias to fuzzer evaluations (see Section 4).

2 THE FUZZTASTIC COVERAGE ANALYZER

This section describes FuzzTastic, which addresses the problem of reliably collecting fine-grained coverage data. Thereby, we first describe the steps and software artifacts required to set up FuzzTastic, followed by a detailed overview of how our tool works.

2.1 Program Analysis and Instrumentation

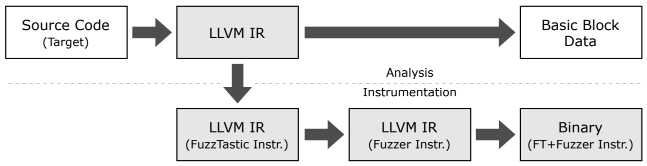

Figure 1: Analysis and instrumentation artifacts.

Figure 1 shows the analysis and instrumentation artifacts involved when using FuzzTastic. First, the source code of the program to be fuzzed is translated into LLVM's intermediate representation (IR). The respective IR file is then used to analyze and instrument each basic block as described below.

```
"basic_blocks": [
  {
    "id": 0,
    "file": "gif2png.c",
    "function": "interlace_line",
    "lines": [45, 48, 50]
  }, ...
]
```

```
"timestamp": {
  "start" : "2021-04-22 14:25:19",
  "report": "2021-04-22 16:25:21"
},
"block_coverage": [
  79704586 , ..., 3497484
]
```

Figure 2: Example data of FuzzTastic.

Static Analysis. Using the generated IR file, we run a custom LLVM compiler pass to extract metadata from the individual basic blocks that comprise the following information:

- **Identifier:** Unique numerical identifier (ID) of the basic block within the program.
- **File:** Path to the C/C++ source file in which the basic block is located.
- **Function:** Name of the function whose control-flow graph (CFG) contains the basic block.
- **Lines:** List of source code line numbers (comments excluded) enclosed by the basic block.

The left part of Fig. 2 provides an example of such metadata (JSON format) of the program Gif2png. Therein, the basic block with ID 0 covers the lines 45–50 in function `interlace_line`, which in turn is defined in the source file `gif2png.c`.

[3] On Google Cloud instances, generating such a dataset would cost over $2,000 [15].

```
static int32_t ft_shm_data[<N_BASIC_BLOCKS>];

void __ft_inc_cov(int32_t bb_id) {
  ft_shm_data[bb_id] += 1;
}
```

Figure 3: FuzzTastic instrumentation.

Instrumentation. First, we use the IR file, initially generated for the static program analysis, to apply FuzzTastic's instrumentation in which each basic block is extended with a call to the function `__ft_inc_cov` (see Fig. 3) stored in a dynamic runtime library. Accordingly, whenever a basic block is executed, this function is called and increments the hit-count of the respective block in the array `ft_shm_data`. Since we measure the coverage of all executed inputs generated during the fuzzing campaign, and not only those stored in the queue of the fuzzer (as FuzzBench does), our approach is not affected by path collisions existent in most coverage-based fuzzers. Note that `ft_shm_data` is stored in a shared memory segment such that the coverage information can be queried by FuzzTastic parallel to the fuzzing campaign, which is less time and resource consuming than replaying the inputs in the queue. After that, we run the instrumentation required by grey- and whitebox fuzzers, followed by compiling the instrumented IR file into a fuzzable binary.

2.2 Coverage Analysis

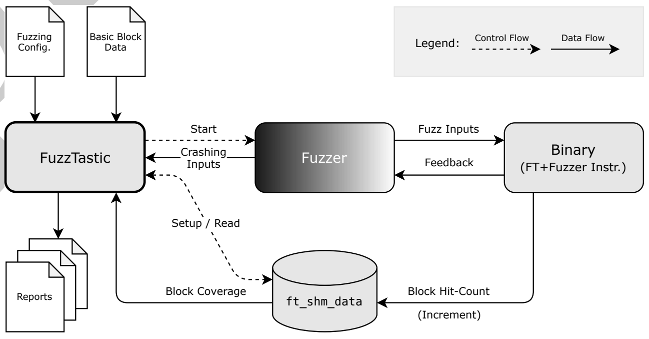

Figure 4: Big picture of FuzzTastic.

Figure 4 shows the interactions, *i.e.*, the control- and data-flow, between FuzzTastic, the chosen fuzzer (black-, grey-, or whitebox), and the instrumented target program.

Configuration. As input, FuzzTastic takes the extracted basic block metadata of the target program and a user-defined configuration (bash scripts) of the fuzzing campaign. These scripts specify the path to the target program plus the command-line interface (CLI) options with which to execute it. They also contain the path to the fuzzer and its CLI options *resp.* environment variables. Furthermore, the configuration specifies the fuzzer's working directory, the directory with the initial seeds, and the fuzzing timeout. Lastly, it sets the directory where the coverage reports should be written along with the interval (default: 15 minutes) at which the reports should be persisted. Accordingly, to customize FuzzTastic to other programs or fuzzers, only those scripts need to be modified.

Table 1: Benchmark programs

Subject	Version	Driver	LoC	# Blocks	# Funcs.
		`nm`	68,667	44,114	2,126
		`objdump`	89,961	60,448	2,701
Binutils	2.29	`readelf`	22,347	18,578	477
		`size`	68,115	43,711	2,101
		`strings`	68,048	43,714	2,093
FFmpeg	n3.3.2	`ffmpeg`	522,813	432,244	21,147
FreeType2	2.7	`ftfuzzer`	44,686	27,521	1,635
Gif2png	2.5.3	`gif2png`	988	700	27
JasPer	1.900.0	`jasper`	17,385	14,417	720
JsonCpp	1.8.4	`jsoncpp_fuzz` [1]	7,251	5,938	1,328
Libpcap	1.9.0	`fuzz_pcap`	12,076	6,442	497
Zlib	1.2.9	`zlib_fuzzer` [1,2]	4,223	3,289	148
Total			926,560	701,116	35,000

[1] Fuzz driver provided by OSS-Fuzz

[2] Original name: `zlib_uncompress_fuzzer`

Coverage Measurement. After setting up the shared memory segment and starting the fuzzing campaign, FuzzTastic repeatedly reads in real-time the coverage information and stores them into separate coverage reports until the timeout has expired. These reports contain the following data:

- **Timestamp:** Time when the campaign was started (`start`) and when the coverage data was recorded (`report`).
- **Block Coverage:** List containing the exact hit-counts of each basic block, with the list indices being aligned to the block IDs.

An example of such a coverage report, generated after 2 hours of fuzzing Gif2png with the AFL fuzzer, can be seen on the right side of Fig. 2. Within this period, the coverage data indicates that the basic block with ID 0 has been executed 79,704,586 times, while the least frequently executed block counts only 2,353 executions. Note that besides basic block coverage, the generated reports also allow measuring function coverage and statement coverage (subsumed by block coverage [21]). In future versions of FuzzTastic, we plan to output CFG information about the subject's functions to also support edge *a.k.a.* branch coverage.

Limitations. This in-depth, path-collision-free coverage measurement comes with the limitation that programs exceeding a certain size cannot be analyzed with FuzzTastic, as the array `ft_shm_data` (see Fig. 3) would require too much random-access memory (RAM). However, as shown in the next section, even large-scale programs like FFmpeg, which contains a total of 432,244 basic blocks, can be easily analyzed with only 4GB RAM. Also, similar to other coverage analyzers, the code instrumentation incurs an extra overhead in terms of binary size and performance. By instrumenting at the granularity of basic blocks (instead of instructions) and using efficient shared memory based inter-process communication, we tried to keep the overhead as low as possible.

3 GENERATING A BENCHMARK DATASET

This section describes the experimental setup used to create a benchmark dataset with FuzzTastic. In total, this dataset contains in-depth coverage data from ~12 CPU-years of fuzzing, *i.e.*, 12 subject programs × 9 fuzzers × 2 different initial seeds (empty/non-empty) × 20 repetitions × 24 hours.

Subject Programs. For our benchmark dataset we selected a diverse set of 12 different free and open-source (FOS) C/C++ applications and libraries (see Table 1). We based our program selection on two criteria. First, to create a setup close to real-world fuzzing, we chose open-source programs that are widely and actively used in practice. For libraries, we made sure that a fuzz driver is provided either by the developers themselves or by OSS-Fuzz [19]. Second, we focused on selecting programs from different domains to enable more generalizable evaluations. Here, the selection ranges from programs for binary manipulations (Binutils), over data processors and converters for different file formats (FFmpeg, FreeType2, Gif2png, JasPer, and JsonCpp), to network utilities (Libpcap) and data compression tools (Zlib).

Fuzzers. As fuzzing tools, we selected 9 different FOS coverage-based greybox fuzzers that have been published at top-tier research venues and/or are popular and widely used among practitioners, using GitHub stars as an indicator [6] for this.

One of the most widespread fuzzers is AFL [1] (version: 2.56b). It implements an evolutionary fuzzing technique that assigns energy to seeds based on their branch coverage, execution time, and discovery time of coverage-increasing inputs. AFLFast [5] (2.51b) extends AFL with an enhanced power schedule algorithm that assigns energy to seeds based on high-/low-frequency program paths executed using a Markov chain model of basic block transition probability. Another employed AFL variant is AFL++ [8] (2.64c), a community-driven tool that incorporates proven fuzzing techniques proposed in research papers such as AFLFast's seed power schedules and MOpt's fuzz mutators (discussed below). AFLSmart [18] (2.52b) is also built on top of AFL, introducing structural fuzzing by modifying the higher-level representation of a seed instead of its raw bytes. We also use Eclipser [7], which implements a concolic testing engine that uses an approximation of path constraints resolvable by generational search, as opposed to costly SMT solving. Furthermore, we include FairFuzz [12] (2.52b) which is also based on AFL. It implements novel mutators that focus on generating fuzz inputs that exercise branches which guard empirically hard to cover source code. Another employed fuzzer is Honggfuzz [3] (2.2), which supports low-level process tracing, making it possible to intercept hijacked signals from crashes that otherwise are often concealed by the fuzzed program. Lastly, we include the fuzzers MOpt-AFL (2.52b) and MOpt-AFL++ (2.64c), which utilize an optimized mutation scheme based on a customized particle swarm optimization algorithm, called MOpt [13], to find the optimal strategy for selecting fuzz mutators.

Seeds, Timeout, and Repetitions. We execute each {subject × fuzzer}-pair with the empty and a non-empty initial seed inputs to be able to study their effects on the fuzzing performance. Regarding the empty seed, we use the smallest amount of data so that it is not rejected by the subject's input parser, *e.g.*, `{}` for JsonCpp. For the non-empty seeds, we take the provided seeds from AFL's GitHub repository. To include performance peaks of fuzzing techniques that increase their effectiveness later in the campaign, *e.g.*, when more seed inputs are added to the queue, we chose a time limit of 24 hours for all experiments. Moreover, we repeat each experiment 20 times so that future evaluations can account for the randomness involved in fuzzing through statistical tests.

Infrastructure. We run all experiments on a high-performance computing (HPC) cluster with Intel® Xeon® E5-2690v3 CPU-based nodes and SLES 15 Linux as operating system. Each node counts 28 physical cores, running at a frequency of 2.6GHz, and provides 64GB RAM. Within this HPC cluster, we distribute the fuzzing campaigns across 96 nodes (= 2688 physical cores, 6144GB RAM), where each campaign is assigned one dedicated core and 4GB RAM. Also, to avoid I/O bottlenecks on the file system, we run each fuzzer on a RAM drive.

4 CASE STUDY: PATH COLLISION SEVERITY

In this section, we use the generated dataset to investigate the severity of the path collision problem existent in the widespread AFL fuzzer and its 84[4] affected variants.

Path Collision Problem. This problem [8, 9, 20] is introduced by AFL's code instrumentation for tracking edge coverage. Therein, each basic block $B \in \mathfrak{B}$ in the target program is assigned a numerical random ID $n \in \mathbb{N}$, *i.e.* $B.\text{id} := n$. The function $hash : \mathfrak{B} \times \mathfrak{B} \to \mathbb{N}$, defined as $hash(B_i, B_j) := B_j.\text{id} \oplus (B_i.\text{id} \gg 1)$, is then used to compute a hash value for any given edge $e \in \mathfrak{B} \times \mathfrak{B}$ between two basic blocks. At runtime, the hash of a covered edge e_c serves as key in a shared bitmap `cov` to increment its hit-count, *i.e.* $\texttt{cov}[hash(e_c)]{+}{+}$. However, due to the many basic blocks in real-world programs (see Table 1) and the random numbers of the block IDs, the hashes of two edges e_k and e_l with $k \neq l$ can collide, *i.e.* $hash(e_k) = hash(e_l)$. These collisions prevent the fuzzer from adding coverage-increasing inputs to its queue, which are then also not considered when measuring coverage the FuzzBench-way.

Empirical Evaluation. To show the severity of the path collision problem and to demonstrate FuzzTastic's coverage accuracy (not affected by this problem), we replay all inputs stored in the fuzzer queues of the 3360 fuzzing campaigns (24 hour timeout) run with the 7 AFL-based fuzzers and measure the achieved code coverage. For each fuzzing campaign c, we then compute the missed basic block ratio (MBBR) using the formula:

$$MBBR(c) = \frac{|\mathcal{B}^c_{\text{FuzzTastic}} - \mathcal{B}^c_{\text{Replay}}|}{|\mathcal{B}^c_{\text{FuzzTastic}}|} \tag{1}$$

The set $\mathcal{B}^c_{\text{FuzzTastic}} \subseteq \mathfrak{B}$ contains the basic blocks covered by all fuzz inputs generated during c measured by FuzzTastic and $\mathcal{B}^c_{\text{Replay}} \subseteq \mathcal{B}^c_{\text{FuzzTastic}}$ the blocks covered by the inputs stored in the fuzzer queue within the same campaign. Thereby, to ensure a uniform comparison between both approaches, we use the same code instrumentation for measuring basic block coverage of the replayed queue inputs as for the FuzzTastic analysis tool.

Figure 5 shows the distribution of missed basic block ratios, including the harmonic mean scores (⋄), of the 3360 campaigns on the different subject programs. The red dots thereby indicate the maximal inaccuracy for each subject. In all subjects except for `strings` (Binutils), basic blocks were missed when replaying the inputs from the queue. Moreover, due to the randomness of fuzzing, the percentage of missing blocks within the same subject can vary significantly among different campaigns and thus can also not be pre-estimated. For example, up to 9% of the basic blocks

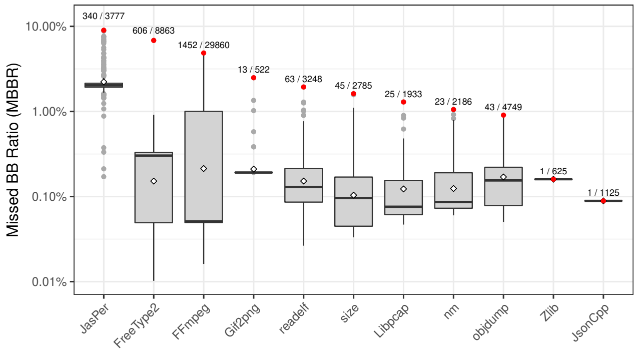

Figure 5: Path Collision Severity

(*i.e.*, 340 out of 3777) were missed in the subject program JasPer. This can lead to biased evaluations, where *e.g.* affected AFL-based fuzzers are rated as less effective in terms of code covered than other, path-collision-free fuzzers. Also note that even a small number of missing basic blocks can lead to misleading results, as some blocks are empirically harder to cover than others. Therefore, fuzzers that manage to trespass these blocks often have to generate exponentially more fuzz inputs [4]. This again underlines the importance of accurately measuring code coverage in order to ensure correct fuzzer evaluations.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented FuzzTastic, a fine-grained, LLVM-based coverage analyzer that can be attached to any black-, grey-, and whitebox fuzzer. Unlike existing approaches, it captures the exact number of executions per basic block during the fuzzing campaign, which enables frequency-based coverage analysis to evaluate fuzzers from another perspective. Moreover, the coverage data output by FuzzTastic can be used to visualize fuzzing progress[5], *e.g.*, the functions covered over time and across multiple trials. Furthermore, we created a large-scale benchmark dataset using FuzzTastic that enables fuzzer evaluations that are not biased by the largely ignored path collision problem of most coverage-based fuzzers such as AFL. Also, this dataset can be used to detect "roadblocks", *i.e.*, conditions in the code that are not or only rarely passed by the fuzz inputs, in order to improve the heuristic of fuzz mutators. In the future, we plan to incorporate the Magma benchmark [10] to also support bug-based benchmarking.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. Furthermore, we thank the Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities (BAdW) for the provisioning and support of high-performance computing infrastructure essential to this work.

[4] `https://fuzzing-survey.org/` (Accessed: 2021-09-30)

[5] `https://mboehme.github.io/img/freetype2_dot.gif` (Animated call graph created from FuzzTastic data). Therein, the green nodes indicate functions in FreeType2 covered in all 20 trials by Honggfuzz *resp.* AFL, while the orange ones represent functions covered in at least one trial.

REFERENCES

[1] [n. d.]. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/. Accessed: 2021-09-20.

[2] [n. d.]. Clang Documentation: Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html. Accessed: 2021-09-20.

[3] [n. d.]. Honggfuzz: Security-oriented Software Fuzzer. https://honggfuzz.dev/. Accessed: 2021-10-06.

[4] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 713–724. https://doi.org/10.1145/3368089.3409729

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (may 2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841

[6] Hudson Borges and Marco Tulio Valente. 2018. What's in a Github Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[7] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*, Vol. 2019-May. IEEE, 736–747. https://doi.org/10.1109/ICSE.2019.00082

[8] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the USENIX Workshop on Offensive Technologies*.

[9] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, Vol. 2018-May. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040

[10] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* 4, 3 (2021), 81–82. https://doi.org/10.1145/3410220.3456276

[11] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the Conference on Computer and Communications Security*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[12] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[13] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the USENIX Security Symposium*.

[14] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, sang kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[15] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1393–1403. https://doi.org/10.1145/3468264.3473932

[16] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of Unix Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

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

[18] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2020. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2020), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

[19] Kostya Serebryany. 2017. OSS-Fuzz: Google's Continuous Fuzzing Service for Open-Source Software. USENIX Association, Vancouver, BC.

[20] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses* (2019), 1–15.

[21] Elaine J. Weyuker. 1986. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering* SE-12, 12 (dec 1986), 1128–1138. https://doi.org/10.1109/TSE.1986.6313008