SBFT Tool Competition 2023 - Fuzzing Track

Dongge Liu* Google, USA
Jonathan Metzman* Google, USA
Marcel Böhme† MPI-SP, Germany Monash, Australia
Oliver Chang* Google, USA
Abhishek Arya* Google, USA

*{donggeliu, metzman, ochang, aarya}@google.com
†marcel.boehme@mpi-sp.org

2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT) | 979-8-3503-0182-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBFT59156.2023.00016

***Abstract*—This report outlines the objectives, methodology, challenges, and results of the first Fuzzing Competition held at SBFT 2023. The competition utilized FUZZBENCH to assess the code-coverage performance and bug-finding efficacy of eight participating fuzzers over 23 hours. The competition was organized in three phases. In the first phase, participants were asked to integrate their fuzzers into FUZZBENCH and allowed them to privately run local experiments against the publicly available benchmarks. In the second phase, we publicly ran all submitted fuzzers on the publicly available benchmarks and allowed participants to fix any remaining bugs in their fuzzers. In the third phase, we publicly ran all submitted fuzzers plus three widely-used baseline fuzzers on a hidden set and the publicly available set of benchmark programs to establish the final results.**

***Index Terms*—fuzzing, evaluation, open-source.**

I. INTRODUCTION

We report on the organization of the first fuzzing competition at the 16th International Workshop on Search-Based and Fuzz Testing (SBFT) held on the 14th of May 2023 in Melbourne, Australia. The objectives of this competition were (i) to evaluate the performance of the fuzzers submitted to this competition in terms of coverage and bug finding ability, (ii) to gather experience and feedback on the sound benchmarking of fuzzing tools, and (iii) to stress test the FUZZBENCH benchmarking platform which has been built particularly for this purpose.

Throughout the competition we paid particular attention to the mitigation of different forms of bias. For instance, in order to avoid overfitting to a particular set of benchmarks (confirmation bias), we allowed participants to develop, integrate, and evaluate their fuzzers privately on a publically available set of benchmarks while conducting the actual competition on a set of benchmarks that included a large number of hidden benchmarks. In order to avoid survivorship bias, we do *not* evaluate their bug finding ability on a *given* set of bugs that we already know how to find. Instead, we evaluate their bug finding ability in terms bugs found by any fuzzer. We make sure to use the same AddressSanitizer (ASAN) instrumented binaries across all fuzzers.

In summary, we found that the AFLRUSTRUST fuzzer performed well in terms of both, the coverage achieved and bugs found. The fuzzers LIBAFL_LIBFUZZER, HASTEFUZZ, and AFL+++ excelled on coverage-based benchmarks, while PASTIS and AFLSMART++ found more bugs than the average fuzzer. We present the final ranking and more concrete results live at the tool competition.

II. FUZZBENCH: FUZZER BENCHMARKING PLATFORM

FUZZBENCH [1] is a free, open source fuzzer benchmarking service built to make fuzzer benchmarking easy and rigorous. It allows researchers, who are interested in evaluating their fuzzers against other state-of-the-art fuzzers, to launch large-scale experiments in a free and reproducible manner.

The FUZZBENCH infrastructure consists of a large number of publicly available benchmark programs taken from OSS-FUZZ[1]. The benchmark programs are open source C/C++ programs carefully integrated by their maintainers, and include programs like `Curl`[2], `OpenSSL`[3], `PHP`[4], and `systemd`[5]. Because the source code for most FUZZBENCH experiments is made public and the specific FUZZBENCH version can be pinned, reproducing FUZZBENCH experiments is often much easier than reproducing bespoke experiments used in other research.

FUZZBENCH can conduct bug-based or code coverage-based experiments [2]. Throughout out the course of an experiment, and upon its completion, FUZZBENCH generates a report detailing the performance of each fuzzer. The report compares fuzzers based on their performance across all benchmarks as well as on individual benchmarks and shows effect size (Vargha Delaney $\hat{A}_{12}$) and statistical significance (Mann Whitney U test). The comparison across all benchmarks contains two rankings, one based on their average rank on each individual benchmark and one based on their performance relative to the best performing fuzzer on each individual benchmark. FUZZBENCH reports include a critical difference diagram so that users can see if differences between fuzzers based on average rank is statistically significant. The report's comparison on individual benchmarks consists of graphs and data showing, the number of crashes found and the growth of code coverage throughout the experiment.

[1]https://google.github.io/oss-fuzz/
[2]https://github.com/curl/curl
[3]https://github.com/openssl/openssl
[4]https://github.com/php/php-src
[5]https://github.com/systemd/systemd

To request an experiment, the interested researcher submits a pull request to the Github repository where the fuzzer is integrated or privately emails fuzzbench@google.com. A typical experiment in FUZZBENCH involves about 20 trials of 10 fuzzers running on 20 benchmarks for 23 hours. This is about 10-CPU years, which is cost prohibitive for most researchers. Researchers can use FUZZBENCH by integrating with a simple Python and Docker based API. This integration usually is less than 100 lines of code.

FUZZBENCH has had an enormous impact on fuzzer development and research. Over 900 experiments have been conducted using the FUZZBENCH service. FUZZBENCH has been discussed in over 100 academic papers. And FUZZBENCH has been used to guide the development of popular fuzzers such as AFL++, HONGGFUZZ and LIBFUZZER. FUZZBENCH experiments have most desirable qualities that Klee et al. [3] described most evaluations as lacking, including: statistically sound comparisons and statistical tests, long timeouts and real-world programs.

III. COMPETITION SETUP

Phases. The competition was organized in three phases. In the first phase, participants were asked to integrate their fuzzers into FUZZBENCH and allowed them to privately run local experiments against the publicly available benchmarks. In the second phase, we publicly ran all submitted fuzzers on the publicly available benchmarks and allowed participants to fix any remaining bugs in their fuzzers. In the third phase, we publicly ran all submitted fuzzers plus three widely-used baseline fuzzers on a hidden set and the publicly available set of benchmark programs to establish the final results.

Performance metrics. In our competition, we measure both the code coverage achieved and the bug-finding capacity to compare the performance of the submitted fuzzers [3], [4]. As benchmarking platform, we use FUZZBENCH which measures line coverage across all coverage-based benchmarks and the time it takes to generate the first crashing input across all bug-based benchmarks. To facilitate a more intuitive comparison of fuzzer performance in both categories, we present a *relative median score* for each fuzzer.

We compute the coverage-based score for each fuzzer as follows. As it is impractical to determine the total number of reachable lines in each coverage-based benchmark *bc* [5], we compute the relative coverage score $score(bc, f)$ for a fuzzer f by dividing the median value of its line coverage over 20 trials (i.e., $\mathrm{cov}(bc, f, n)$ where $n = 1..20$) by the maximum line coverage attained by all fuzzers F on that specific benchmark:

$$score(bc, f) = \frac{\mathrm{cov}(bc, f)}{\max\limits_{i \in F} \max\limits_{n=1..20} \mathrm{cov}(bc, i, n)} \tag{1}$$

$$\mathrm{cov}(bc, f) = \underset{n=1..20}{\mathrm{Med}}(\mathrm{cov}(bc, f, n)) \tag{2}$$

We compute the bug-based score for each fuzzer as follows. Many fuzzer-generated crashing inputs may expose the same bug, and the same bug may yield different stack traces [6], [7]. In order to circumvent challenges of bug deduplication, we include only one reproducible bug in each benchmark and measure the time it takes to generate the first input that causes the benchmark binary to crash. Therefore, considering that each bug-based benchmark bb comprises only one bug, we calculate the relative score $score(bb, f)$ of a fuzzer f using the following method:

$$score(bb, f) = \underset{n=1..20}{\mathrm{Med}}(\mathrm{bug}(bb, f, n)) \tag{3}$$

$$\mathrm{bug}(bb, f, n) = \begin{cases} 1 & \text{if } f \text{ finds a bug in } bb \text{ in trial } n \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

In instances where multiple fuzzers detect an equal number of bugs across all benchmarks, we additionally provide their average time required for bug discovery as an auxiliary metric.

Benchmarks. The 53 benchmarks employed in this study were selected from a diverse range of real-world open-source projects integrated into OSS-FUZZ. This approach ensures that researchers can evaluate their fuzzers on the latest, popular, and actively maintained real-world open-source programs. Meanwhile, project maintainers can benefit from state-of-the-art fuzzers.

To guarantee the reproducibility of fuzzer performance, each benchmark is anchored to a specific commit. In particular, the commit for each bug-based benchmark are carefully chosen such that the bug present have been fixed or published within one year. This approach prevents security vulnerability leakage while maintaining benchmarks up-to-date for research evaluation purposes.

Benchmarks are divided into public and private sets. The *public benchmark set*, consisting of 5 bug-based and 24 coverage-based benchmarks, is made available to participants for build and runtime errors identification upon joining the competition. In contrast, the *private benchmark set*, comprising of 10 bug-based and 14 coverage-based benchmarks, is withheld until the final evaluation to mitigate overfitting.

Preventing overfitting in fuzzing competitions is typically challenging since participants usually require access to the benchmark source code to identify and resolve compatibility issues. However, FUZZBENCH's design effectively addresses this issue by separating the benchmarks and fuzzers. This allows fuzzers to be built and run on private benchmarks using the same code that was tested on the public ones, contributing to a fair and impartial evaluation of fuzzer performance.

Fuzzers. The competition evaluates a total of 12 fuzzers, including 8 fuzzers submitted by participants and 4 fuzzers used as baseline. The participant-submitted fuzzers are AFL+++[6], AFLRUSTRUST[7], AFLSMART++[8], HASTEFUZZ[9], LEARNPERFFUZZ[10], LIBAFL_LIBFUZZER[11], PASTIS[12], and SYM-

[6]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/aflplusplusplus
[7]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/aflrustrust
[8]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/aflsmart_plusplus
[9]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/hastefuzz
[10]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/learnperffuzz
[11]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/libafl_libfuzzer
[12]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/pastis

SAN[13]. The four baseline fuzzers encompass AFL[14], AFL++[15], HONGGFUZZ[16], and LIBFUZZER[17]. We selected AFL and AFL++ as baselines, as most participants extended them to construct their own. The fuzzers HONGGFUZZ and LIBFUZZER were chosen due to their contribution to the discovery of bugs in the bug-based benchmarks under OSS-FUZZ production environment.

Platform and Configuration. The competition is conducted on `Google` Cloud virtual machines. We concurrently measure 20 trials per fuzzer on each benchmark, with each trial executing one fuzzer instance on one benchmark. Each trial was run on a dedicated clean `Ubuntu20.04` virtual machine instance equipped with 1 vCPU and 3.75 GB memory. For some benchmarks, seed corpora were available, mirroring the production environment in OSS-FUZZ.

IV. EVALUATION RESULTS

We present and discuss the results of coverage-based and bug-based benchmarking separately. From previous experiments [4], we do not expect a strong agreement between rankings established by coverage-based versus bug-based benchmarking, but they each provide important and interesting insights about the capabilities of the fuzzers.

A. Coverage-based Benchmarking

We first focus on the fuzzers' ability to cover the most code possible. Bugs cannot be found in code that is not covered.

We find that LIBAFL_LIBFUZZER leads in 23 out of 38 coverage-based benchmarks, significantly more than any other fuzzer. However, its overall performance is negatively impacted by the near-zero coverage exhibited on three benchmarks: `draco`, `dropbear`, and `proj4`. In particular, LIBAFL_LIBFUZZER generated merely two input cases for `draco` and crashed immediately after initiating `dropbear`. To facilitate debugging, FUZZBENCH has provided researchers with the input corpora and fuzzer logs.

HASTEFUZZ consistently performs well on all coverage-based benchmarks, securing its position as one of the best fuzzers. Although its relative median scores ranked first on only 16 benchmarks, it remained within the 90% relative median range on 31 benchmarks and secured a position within the top three on 35 benchmarks. Notably, it exhibited the lowest standard deviation across all benchmarks (approximately 8.15), which is less than half of the second-lowest (AFL+++, 17.61).

Both AFL+++ and AFLRUSTRUST display competitive performance across the majority of benchmarks. Their relative scores ranked first on 16 and 12 benchmarks, respectively, achieved within the 90% range on 31 and 29 benchmarks, and secured top three positions on 33 and 22 benchmarks.

As for baseline fuzzers, AFL++ emerged as the best-performing and outperformed most other fuzzers on the majority of benchmarks. Its average relative score is 92.67, whereas the highest average score among the remaining participant fuzzers is below 90.

A notable observation is that many top-performing fuzzers exhibit a high degree of similarity in their coverage performance, primarily due to their shared underlying fuzzer architecture. To measure "coverage similarity", we consider the coverage achieved by two fuzzers across different benchmarks and compute the cosine similarity. We find that the cosine similarity between AFLRUSTRUST and AFL+++ surpasses 0.99, signifying their nearly identical relative median scores across all benchmarks. Likewise, the cosine similarities among AFLRUSTRUST and HASTEFUZZ, HASTEFUZZ and AFL++, AFL++ and AFL+++ are all above 0.98. In contrast, the cosine similarities between LIBFUZZER and AFL++, LIBFUZZER and AFLRUSTRUST, LIBFUZZER and AFL+++ are approximately 0.93.

Our analysis reveals that certain benchmarks are adept at distinguish the coverage performance of fuzzers. For instance, after excluding outliers, the `openthread` benchmark exhibits the highest interquartile range of 22.25, along with a standard deviation of 18.91. The range of fuzzer scores on this benchmark spans from 98 to 49, indicating that the top-performing fuzzer achieves approximately double the relative coverage of the lowest-performing one.

Similarly, the scores on the `lcms` benchmark range from 95 to 19, yielding a standard deviation of 22.79 and an interquartile range of 18.50. For the `freetype2` benchmark, the standard deviation is 19.98, with an interquartile range of 21.75 and fuzzer scores ranging from 22 to 95. Furthermore, no fuzzer's relative median score exceeds 68 on the `botan` benchmark, suggesting that the maximum of median scores of all fuzzers is approximately two-thirds of the highest line coverage across all trials.

Conversely, some benchmarks display a high degree of similarity in performance across fuzzers, thereby offering limited utility in differentiating and ranking them. For example, all fuzzers are within 98% of the top-performing fuzzer's score on the `libjpeg` benchmark, and almost all of them achieve the same line coverage on the `firestore` benchmark.

B. Bug-based Benchmarking

In terms of bug finding, many fuzzers display similar performance on bug-based benchmarks. For instance, AFLRUSTRUST and PASTIS both have the highest relative median score (53.33), indicating that their median-performing fuzzer trials covered 8 out of 15 bugs across all benchmarks. Likewise, participant-submitted fuzzers AFL+++ and HASTEFUZZ covered 6 bugs, equal to the performance of baseline fuzzers AFL++ and LIBFUZZER.

Seven benchmarks were found to be particularly useful in differentiating fuzzers in this competition, as they exhibited diverse performance among fuzzers: `aspell`, `assimp`, `file`, `bloaty`, `ffmpeg`, `libaom`, and `libxml2`. Both

[13]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/symsan
[14]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/afl
[15]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/aflplusplusff
[16]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/honggfuzz
[17]https://github.com/google/fuzzbench/tree/SBFT'23/fuzzers/libfuzzer

AFLRUSTRUST and PASTIS discovered 6 out of the 7 bugs in these benchmarks, outperforming other fuzzers. However, AFLRUSTRUST and PASTIS had slightly different bug-finding patterns; AFLRUSTRUST covered a comparatively rare bug in `file` but missed a more commonly found bug in `ffmpeg`.

While half of the fuzzers found more than 4 bugs overall, the symbolic-based fuzzer SYMSAN discovered only 1 bug in `assimp`. Interestingly, LIBAFL_LIBFUZZER, which performed well across coverage-based benchmarks and found bugs in 5 benchmarks, was the only fuzzer that missed the bug in `assimp`. This result could be attributed to its relatively low coverage on this specific benchmark.

We also examined the average time required for fuzzers to discover a bug. PASTIS proved to be the fastest in detecting bugs on average, with AFLRUSTRUST and AFLSMART++ following closely behind. Notably, the cosine similarity between AFL++ and AFL+++ exceeds 0.98, suggesting that they frequently identify bugs at approximately the same time. Likewise, the cosine similarity between HONGGFUZZ and PASTIS surpasses 0.9, indicating a comparable speed in causing crashes within the benchmark. LIBAFL_LIBFUZZER appears to possess a distinct design, resulting in the lowest similarity score when compared to any other fuzzers.

The bug-based benchmarks in this competition also underscore the "asymmetry" between coverage-based and bug-based rankings, as highlighted by Böhme et al. [4]. For instance, HASTEFUZZ excelled in coverage-based benchmarks yet discovered fewer bugs. Conversely, AFL identified more bugs than AFL++, despite covering less code. Although code coverage is a well-established and easily measurable benchmarking metric, these findings stress the significance of taking bug-finding capabilities into consideration when optimizing for higher coverage and evaluating fuzzers. Essentially, fuzzers are intended to detect bugs, with coverage serving as a heuristic to estimate their bug-finding potential.

Bug-based benchmarking presents several challenges that we tackled in different ways. Firstly, acquiring the source code of real-world bugs is arduous, and the performance measured by artificial bugs might not accurately reflect reality. FUZZBENCH addresses this issue by using bugs filed by OSS-FUZZ when fuzzing actual open-source projects, providing a ground truth for bugs that had been and need to be discovered in production.

Secondly, a systematic approach for selecting appropriate bug benchmarks for evaluation remains absent. For instance, if all fuzzers exhibit similar performance on certain benchmarks, those bugs offer limited value into fuzzer assessment. To mitigate this concern, we incorporated benchmarks that were hidden during development and only revealing during final evaluation, culminating in nine benchmarks that demonstrate varying bug-discovery performances among fuzzers in this competition.

Thirdly, determining the superior fuzzer performance becomes difficult when multiple fuzzers can discover the same bug. To address this, we employ an auxiliary metric, i.e., measuing the average time required by each fuzzer to discover a bug. While FUZZBENCH evaluates this metric at 15-minute intervals, which may occasionally compromise accuracy, we highlight that this potential risk does not unfairly benefit any specific fuzzer.

Finally, ascertaining whether multiple crashes correspond to the same bug by grouping backtraces poses a considerable challenge. To tackle this issue, the competition restricts each benchmark to include only one known bug. Each associated open-source project is subjected to rigorous testing using multiple fuzzers over an extended period to minimize the likelihood of multiple reproducible bugs coexisting within a single benchmark.

V. CONCLUSION AND FUTURE WORK

In this competition, FUZZBENCH evaluates participant fuzzers and common baselines, comparing them using a variety of statistical tools. The assessment encompasses two key metrics: code coverage and bug-finding. Benchmarks for both metrics are derived from real-world open-source projects, and all fuzzers are tested under uniform production-like environment.

Moving forward, FUZZBENCH aims to enhance the statistical analysis by providing more detailed information, particularly concerning lines or bugs that fuzzers failed to cover. Additionally, FUZZBENCH plans to incorporate a larger collection of bug-based benchmarks to facilitate more comprehensive statistical reasoning.

REFERENCES

[1] 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, pp. 1393–1403.

[2] D. Asprone, J. Metzman, A. Arya, G. Guizzo, and F. Sarro, "Comparing fuzzers on a level playing field with fuzzbench," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 302–311.

[3] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[4] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1621–1633.

[5] D. Liyanage, M. Böhme, C. Tantithamthavorn, and S. Lipp, "Reachable coverage: Estimating saturation in fuzzing," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, 2023, pp. 1–13.

[6] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1084–1093.

[7] I. M. Rodrigues, A. Khvorov, D. Aloise, R. Vasiliev, D. Koznov, E. R. Fernandes, G. Chernishev, D. Luciv, and N. Povarov, "Tracesim: An alignment method for computing stack trace similarity," *Empirical Software Engineering*, vol. 27, no. 2, p. 53, 2022.