FuzzBench: An Open Fuzzer Benchmarking Platform and Service

Jonathan Metzman
Google, USA
metzman@google.com

László Szekeres
Google, USA
lszekeres@google.com

Laurent Simon
Google, USA
laurentsimon@google.com

Read Sprabery
Google, USA
sprabery@google.com

Abhishek Arya
Google, USA
aarya@google.com

ABSTRACT

Fuzzing is a key tool used to reduce bugs in production software. At Google, fuzzing has uncovered tens of thousands of bugs. Fuzzing is also a popular subject of academic research. In 2020 alone, over 120 papers were published on the topic of improving, developing, and evaluating fuzzers and fuzzing techniques. Yet, proper evaluation of fuzzing techniques remains elusive. The community has struggled to converge on methodology and standard tools for fuzzer evaluation.

To address this problem, we introduce FuzzBench as an open-source turnkey platform and free service for evaluating fuzzers. It aims to be easy to use, fast, reliable, and provides reproducible experiments. Since its release in March 2020, FuzzBench has been widely used both in industry and academia, carrying out more than 150 experiments for external users. It has been used by several published and in-the-work papers from academic groups, and has had real impact on the most widely used fuzzing tools in industry. The presented case studies suggest that FuzzBench is on its way to becoming a standard fuzzer benchmarking platform.

CCS CONCEPTS

• **Software and its engineering** → *Application specific development environments*; **Software testing and debugging**; • **Security and privacy** → **Software security engineering**; • **Mathematics of computing** → *Hypothesis testing and confidence interval computation*; • **General and reference** → **Evaluation**; **Experimentation**.

KEYWORDS

fuzzing, fuzz testing, benchmarking, testing, software security

ACM Reference Format:
Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473932

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473932

1 INTRODUCTION

Fuzzing has attracted the attention of both industry and academia because it is effective at finding bugs in real-world software, not just in experiments. Today, fuzzing has seen high adoption among developers [34] and is used to find bugs in widely used production software [26, 27, 40, 42]. At Google we have found tens of thousands of bugs [1] with fuzzers like AFL [45], libFuzzer [37] and Honggfuzz [43]. Academic research on fuzzing has driven many improvements since the inception of coverage-guided fuzzing [45] – Google Scholar reports several thousand published papers since 2014 [28].

While fuzzing efforts have been successful in improving software quality, proper evaluation of fuzzing techniques is still a challenge. There is no consensus on which tools and techniques are effective and generalize well for fuzzer comparison. This is in part due to the lack of standard benchmarking tools, metrics, and representative program datasets, all of which have hampered reproducibility [48].

Klees et al. [31] were the first to study the current state of fuzzing evaluations. They analyzed 32 fuzzing research papers and found that none provided enough "evidence to justify general claims of effectiveness". More specifically, some papers do not use a large and diverse set of real-world benchmarks, have too few trials, use short trials, or lack statistical tests. Furthermore, it is hard to cross-compare between all papers as they typically use different evaluation setup and configuration (e.g., how experiments are run and measured), different subjects (benchmark programs) or even different coverage metrics [41].

Another common challenge is that sound fuzzer evaluation has a high cost, both in researcher time and computational resources. A typical evaluation compares a large number of tools on a large number of subjects (benchmark programs). Setting up all these tools and subjects and making sure that each tool-subject pair works together (i.e., compiles, runs) takes significant effort. Some researchers we talked to described spending several months working on evaluation. A sound evaluation also needs massive computation time (on the order of CPU-years) and resources, as each tool-subject pair needs to run multiple times for statistical significance. In practice, it can take up to ~ 11 CPU-years to run a well-conducted experiment (e.g., 24 hours × 20 trials × 10 fuzzers × 20 subjects). On Google Cloud, this experiment could cost over $2,000. Considering the repeated evaluations necessary during the development of a fuzzing tool, research can require CPU-centuries and tens of thousands of dollars.

FuzzBench aims to alleviate these problems by providing an open-source fuzzer benchmarking service. We designed it following

proven best practices [2, 47], with the goal of making fuzzing evaluation easy, fast, rigorous, and reproducible (Section 2). FuzzBench is modular and extensible, so integrating a fuzzer is easy and requires about 100 LoC on average (Section 3). Even complex fuzzers such as SymCC can be integrated into FuzzBench in 100 LoC [7].

The FuzzBench service uses Google's compute resources. Researchers can request a new evaluation experiment for free (Section 4). By default, the service runs a large-scale experiment with 20 trials of 23 hours each, on (> 20) real-world benchmarks (Section 2). FuzzBench generates reports with statistical confidence intervals and makes the raw data available for further analysis if needed. FuzzBench allows researchers to focus more of their time on perfecting techniques and less time on setting up evaluations and troubleshooting other fuzzers they want to compare against.

There is prior work on designing fuzzing benchmarks, most notably Google's fuzzer-test-suite [38], LAVA [20], Magma [30] and UNIFUZZ [32]. Google's fuzzer-test-suite can be considered the precursor of FuzzBench as FuzzBench initially used many of the same benchmark programs. LAVA and Magma provide a set of programs containing bugs (artificial and real-world, respectively). They measure only the number of bugs found, without measuring code coverage. FuzzBench can use either code coverage or bug discovery for evaluation. Many fuzzing papers follow this practice [6, 9].

To the best of our knowledge, FuzzBench is the only fuzzer benchmarking service. FuzzBench's ease of use and techniques such as statistical tests have helped numerous researchers raise the bar for state-of-the-art fuzzing. Since we released it in 2020 [35], it has been actively used at Google and by the broader fuzzing community both in industry and academia [9, 36]. The two main use cases of FuzzBench today are (1) comparing different fuzzing tools, and (2) determining the effect of a potential improvement or feature (i.e., comparing to a prior version of the same tool). Fuzzer comparison is used both by researchers and by fuzzer users to determine which tool might be best for them. For example, OSS-Fuzz decided to replace AFL with AFL++ based on AFL++'s [22] performance on FuzzBench. In the second use case, developers iterate on their tools by carrying out A/B tests with FuzzBench for a given change and determine the change's impact on fuzzing effectiveness. Popular fuzzing tools in industry – such as libFuzzer, AFL++ and Honggfuzz– use FuzzBench continuously to drive development this way.

As an example of the first use case, FuzzBench periodically evaluates and compares the latest versions of the most commonly used fuzzing tools. We discuss the latest result of this experiment in Section 5. We also evaluate our choices of the parameters used for this main experiment (such as experiment length and other configurations) in Section 6, and share the insights we have gained. We present several case studies for the second use case as well, showing how FuzzBench is being used to successfully improve fuzzing tools (Section 7), and discuss lessons we have learned.

In summary, we make the following contributions:

- We describe the design and implementation of FuzzBench: the first scalable, modular, fuzzer benchmarking-as-a-service platform.
- We present the results of our large-scale experiment on widely-used fuzzers.
- We evaluate our choices of the default experiment parameters (duration, set of benchmarks, seed corpus, etc.).
- We report on case studies, from across academia and industry, on how FuzzBench was used to drive fuzzer development.

2 BENCHMARK METHODOLOGY

The benchmarking methodology of FuzzBench follows best practices gathered over many years at Google through evaluation of fuzzing tools we develop and use [37, 43, 45], and best practices from academic research [2, 31, 47].

2.1 Real-World Benchmark Programs

To evaluate how a tool works on real-world software, we need to use real-world software as benchmark programs. FuzzBench uses OSS-Fuzz [3] projects and their fuzz targets as benchmarks. OSS-Fuzz is a community fuzzing service that continuously fuzzes open source projects. Today there are approximately 400 open source projects fuzzed by OSS-Fuzz. Any OSS-Fuzz fuzz target can be easily added to FuzzBench as a benchmark. We picked a large, diverse, and representative set of fuzz targets from OSS-Fuzz (shown in Table 1) as a default set of benchmarks in FuzzBench. We recommend using this default set for a fair evaluation as these programs span a wide range of applications and coding styles.

The default set of benchmarks was chosen with a particular challenge in mind: how to ensure researchers don't optimize their tools in ways that won't be broadly applicable beyond the particular evaluation. The set contains 22 different benchmarks that process a wide variety of input formats, and are some of the most commonly used open source projects. We believe that even if a tool is optimized to do well on this set of benchmarks, it will likely do well in general due to the size and diversity of the benchmarks.

Table 1: Benchmark programs.

Benchmark	Dictionary	Format	Seeds	Edges
bloaty_fuzz_target	✗	ELF/DWARF/Mach-O	94	89530
curl_curl_fuzzer_http	✗	HTTP response	31	62523
freetype2-2017	✗	TTF, OTF, WOFF	2	19056
harfbuzz-1.3.2	✗	TTF, OTF, TTC	58	10021
jsoncpp_jsoncpp_fuzzer	✓	JSON	0	5536
lcms-2017-03-21	✓	ICC profile	1	6959
libjpeg-turbo-07-2017	✗	JPEG	1	9586
libpcap_fuzz_both	✗	PCAP	0	8149
libpng-1.2.56	✓	PNG	1	2991
libxml2-v2.9.2	✓	XML	0	50461
mbedtls_fuzz_dtlsclient	✗	custom	1	10942
openssl_x509	✓	DER certificate	2241	45989
openthread-2019-12-23	✗	custom	0	17932
php_php-fuzz-parser	✓	PHP	2782	12376
proj4-2017-08-14	✓	custom	44	6156
re2-2014-12-09	✓	custom	0	6547
sqlite3_ossfuzz	✓	custom	1258	45136
systemd_fuzz-link-parser	✗	custom	6	53453
vorbis-2017-12-11	✗	OGG	1	5022
woff2-2016-05-06	✗	WOFF	62	10923
zlib_zlib_uncompress_fuzzer	✗	Zlib compressed	0	875

2.2 Metrics

Code and bug coverage. FuzzBench measures both code coverage and bug coverage (unique bugs found). However, FuzzBench uses code coverage as its primary evaluation metric for two reasons. A fuzzer can only detect a bug if first it manages to cover the code where the bug is located. The primary challenge fuzzers try to solve is creating inputs that exercise new program states. In fact,

most coverage-guided fuzzers do not even implement any specific "bug detection" capabilities, but rely on independent sanitizers [39], such as ASAN, MSAN, UBSAN. Second, comparing fuzzers based on bug coverage can be misleading since real-world bugs in programs are sparse. Even Magma benchmarks where bugs were manually "forward-ported" have between 7-22 bugs each [30]. This is a small number compared to the many thousands of code locations (e.g., lines/branches) where bugs can be introduced in real-world programs (Table 1). Therefore, if we reported only number of bugs found, without reporting code coverage, it might lead to biased results that do not generalize. This is why we advocate for using the combination of code and bug coverage. Note that beyond code and bug coverage, we allow tool integrators to export their own custom metrics (e.g., number of executions, RAM usage).

Independent code coverage metric. One approach to measuring code coverage is using the coverage reported by each fuzzer being benchmarked. However, this approach is flawed as different fuzzers use different coverage metrics (e.g., basic block, edge, line, etc.). Even if they use the same metric, say edge coverage, different implementations will give different results. For example, AFL uses a fixed size edge counter map, which means that certain edges may be missed due to hash collisions. Another approach is to pick the coverage metric and implementation of one of the fuzzers (e.g., SanitizerCoverage used by libFuzzer) and measure the corpus generated by each fuzzers with that. However this is biased towards the fuzzer whose coverage metric is used. Thus, FuzzBench uses Clang's source-based coverage, an independent coverage implementation that is not used by any fuzzer. Clang's source-based coverage is collision-free, provides easy-to-read coverage reports, and is part of Clang's tooling suite.

Differential coverage. Not only does FuzzBench measure the number of code locations covered and its growth over time, FuzzBench is aware of *which* parts of the code (e.g., lines) each fuzzer covers. FuzzBench uses this information to generate "differential coverage" which it presents in reports. FuzzBench presents this information through coverage reports and graphs that show how many unique regions are found by each fuzzer. The plots show how much code was covered by one fuzzer relative to any other fuzzer (e.g., how much code was covered by libFuzzer and not AFL). They also show how much code was covered by each fuzzer that no other fuzzer covered. This is useful information e.g., to determine how to better combine fuzzers to improve results [29].

2.3 Reproducibility and Version Tracking

Reproducibility is important for a benchmarking platform so that results can be validated. In the FuzzBench source code, fuzzers and benchmarks are pinned to specific versions of that software. FuzzBench reports include the version (the git commit hash) of the FuzzBench source code that was used to produce the experiment. Thus, it is possible to reproduce an experiment by checking out the FuzzBench commit and using the same experiment parameters (e.g., trial duration).

2.4 Reporting and Statistical Tests

To help researchers analyze experiment results, FuzzBench offers several options for experiment reports. The automatically generated default report is easy to understand and provides readers with insight into fuzzer performance across all benchmarks and on individual benchmarks. FuzzBench can also provide alternative, more detailed reports that are easy to customize. All reports make the raw data available so researchers can do their own custom analysis. For custom analyses, researchers can use FuzzBench's analysis library for generating their own plots, tables, and statistical tests [23]. In the following section, we discuss the default report. Past experiment reports are available online at fuzzbench.com and more information about experiment reports is available in the documentation.

To get statistically sound results, we run each fuzzer 20 times for each benchmark. Each of these runs is a "trial", which is 23 hours long by default. We selected these parameters based on prior research guidelines [31]. We show in Section 6 that these default settings are sufficient in practice. The reports show results for each benchmark and "experiment level" results which compare fuzzers across all benchmarks. We run the same types of analyses and generate the same types of plots for code and bug coverage. In the text below, "coverage" means both code or bug coverage, as we generate the same plots/tables for both.

Benchmark-level results. The benchmark-level results in the report show multiple plots, tables and statistical test results for each benchmark. For each benchmark, they contain a plot of the growth of code coverage or bugs discovered over time (aggregated over individual trials), and a box plot of the final coverage distribution (including min, 25%, median, 75%, max). They also contain differential coverage plots mentioned earlier, showing pairwise and globally unique coverage numbers. Finally, they link to a browsable code coverage report for each fuzzer on each benchmark.

As for benchmark-level statistical tests, by default, the report includes pairwise fuzzer tests of effect size and null hypothesis significance. The effect size is determined using the Vargha-Delaney A12 measure and the null hypothesis is rejected with the two-tailed Mann-Whitney U test (example provided in Figure 1), as recommended by Arcuri et al. [4].

Experiment-level results. The report includes "top level" experiment results, comparing fuzzers on *all* benchmarks. The most important of these is a "critical difference diagram". An example is shown in Figure 3. This diagram was introduced by Demsar [14], and is often used in the field of machine learning to compare algorithms over multiple benchmarks (data sets). The diagram compares fuzzers across all benchmarks by visualizing both their average rank and the statistical significance between them. The average ranks are computed based on the medians of the reached coverage of each fuzzer on each benchmark. First it ranks each fuzzer on each benchmark, then averages these rankings across all benchmarks. The groups of fuzzers that are connected with bold lines are not significantly different from each other. The statistical significance is computed using a post-hoc Nemenyi test performed after the Friedman test. To the best of our knowledge, FuzzBench is the only platform that provides holistic "experiment-level" statistical tests.

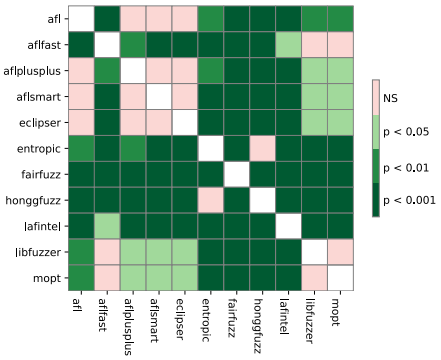

Figure 1: Mann-Whitney U-test result for libxml in fuzzbench.com/reports/paper/Main Experiment. Green cells indicate that the reached coverage distribution of a given fuzzer pair is significantly different.

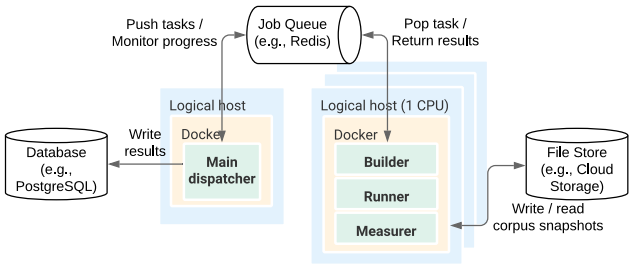

Figure 2: High level-architecture.

Our analysis library offers several alternative cross-benchmark ranking methods. In addition to an "average rank" result, the default report also includes a "top level" result based on "average score". The "average score" defines a fuzzer's score on a given benchmark as the percentage of the highest reached median code-coverage on that benchmark. In the cross-benchmark ranking we take the averages of these scores. While it is unclear which ranking system is "better", we have found both rankings useful in developing and debugging fuzzers (see Section 7).

3 PLATFORM DESIGN

The FuzzBench platform design is divided into two major parts: a user-facing frontend for adding benchmarks and fuzzers, and a backend for running experiments. The frontend provides the interface for benchmark integrations and fuzzer integrations. An important goal of the fronted is to make these integrations easy. Benchmark integrations consist of a `Dockerfile` and bash script (`build.sh`) for each benchmark. Each of these is used to build fuzz targets using the compiler and compiler flags specified by the fuzzer-specific integration. The fuzzer integrations consist of separate `Dockerfiles` for building and running a fuzzer, as well as a `fuzzer.py` script. The `fuzzer.py` uses FuzzBench's API to specify which compiler and compiler flags to use when building benchmarks. In addition, the `fuzzer.py` implements a `fuzz()` function that runs the fuzzer on a specified binary and saves the corpus to a specified directory. Fuzzer integrations consist of modular Python code instead of bash scripts as is common in other platforms. This means integrations can often be reused by similar fuzzers. For example, all AFL-based fuzzers import functionality from AFL's `fuzzer.py` rather than reimplement the same functionality. A typical fuzzer integration can be done in less than 100 lines of code. Once a fuzzer is integrated, it can run any OSS-Fuzz target out of the box.

Important goals of the backend are (1) scalability, (2) fair resource allocation, and (3) platform independence. The high-level architectural design of the backend is shown in Figure 2.

The backend of FuzzBench consists of a dispatcher, and workers for building docker images, running trials, and measuring corpus snapshots. The dispatcher is the most important part of the backend as it is the "brain" of an experiment. When an experiment is started, a dispatcher instance is created, by default, on Google Compute Engine. The dispatcher first spawns jobs to build the docker image for each fuzzer-benchmark pair (by default using Google Cloud Build). During the build stage, the dispatcher also builds a Clang-coverage build of each benchmark that it will use for measuring coverage.

Next, the dispatcher starts Google Compute Engine instances to run trials. In a typical experiment, tens of thousands of Google Compute Engine instances called "runners" are started, which makes experimentation scalable. These instances have one core and 3.75 GB of RAM available. This ensures each fuzzer gets access to the same amount of resources. Each trial runs the docker container of a given fuzzer-benchmark pair, using the `fuzz()` function defined by `fuzzer.py`. While running the fuzzer on the benchmark target, the runner saves the corpus of the fuzzer to Google Cloud Storage at a specific interval (fifteen minutes, by default). The coverage of these corpus "snapshots" is measured by measurer workers to provide results in real-time and provide data on coverage growth throughout the experiment.

The dispatcher starts the measurer workers to measure the coverage of each trial throughout the experiment. Each time a corpus snapshot is saved, a measurer measures its coverage and stores the result in a central SQL database. The dispatcher also periodically regenerates the report based on the results measured so far. This means that a somewhat real-time view of an experiment is available while it is in-progress.

We designed FuzzBench to be as platform independent as possible. This means that not only can users run it on Google Cloud, which some users choose to do, but they can also run experiments locally on their own machines. Local support allows researchers test their tool/benchmark integrations and run small-scale experiments. We are planning to improve support for other cloud platforms and local clusters.

4 THE FUZZBENCH SERVICE

The FuzzBench service works as follows: first a user integrates a fuzzer with the FuzzBench API. This enables the fuzzer to build and fuzz FuzzBench benchmarks (i.e., any OSS-Fuzz target). When submitting a pull request with this integration, the developer also submits an experiment request specifying which fuzzers and benchmarks to benchmark via a YAML file. While the pull request is

reviewed by the FuzzBench maintainers, FuzzBench's continuous integration tests that the fuzzer can build and run every benchmark. Once the pull request is merged, FuzzBench automatically runs the experiment requested by the user. FuzzBench then publishes a report comparing the performance of the fuzzer to other fuzzers. This service-like process is well liked and is used frequently by fuzzer developers, e.g., to improve Honggfuzz [44], libFuzzer [19], and AFL++ [22].

However, public experiment requests present a problem for academic research or other research that cannot be conducted in the open. Academic researchers typically want evaluations of their fuzzer and its source code to be kept private until publication. Therefore we also support private experiments where researchers can send a request to the FuzzBench mailing list (fuzzbench@google.com). In this case, the source code of the fuzzer is kept private, as are the results, which are shared only with the researchers. We store all experiment data in our internal database so that we can make the results public at the time of paper publication to then allow independent analysis and reproduction. Several research groups have used this private service, and some of this research has been published [9, 36].

5 EVALUATION OF POPULAR FUZZERS

In this section we evaluate commonly used fuzzers with FuzzBench. The full results from this experiment (ID: Main Experiment) can be viewed online.[1] In general, the report, data and all details of any experiment referenced in this paper can be viewed by going to fuzzbench.com/reports/paper/<experiment_ID>. For this experiment we benchmarked 11 fuzzers that are important academic works and/or are popular in industry, and are commonly used for comparison in academic papers. Namely, we evaluated AFL, AFLFast AFL++, AFLSmart, Eclipser, Entropic, FairFuzz, Honggfuzz, libFuzzer, MOpt-AFL, and lafintel. Although FuzzBench can benchmark other fuzzers, we have chosen this set to highlight the features of the platform.

We benchmarked the fuzzers on our default benchmark set, containing 22 different open source projects and their fuzz targets from OSS-Fuzz. Table 1 lists the benchmarks and describe some of their characteristics, including dictionary presence, input format, number of seed inputs, and the number of program edges. These benchmarks represent a wide range of userspace programs commonly fuzzed today. They take a variety of input formats, such as XML, JPEG and ELF. There is also variety in whether they come with seed inputs and/or dictionaries. This is useful because not every target that is fuzzed in the real world has dictionaries or seed inputs, as we have learned through OSS-Fuzz [3]. We ran 20 trials for each fuzzer-benchmark pair. Each trial lasted approximately 23 hours. The total runtime was approximately 111,320 CPU hours or a little less than 13 CPU years. We answer a number of research questions (RQ) based on the results.

RQ: Are the evaluated fuzzers significantly different from each other?

The experiment level critical difference diagram is shown in Figure 3. AFL++ came out to be the best, having the highest average rank (3.68), followed by Honggfuzz, Entropic, and Eclipser. Based

[1] fuzzbench.com/reports/paper/Main Experiment

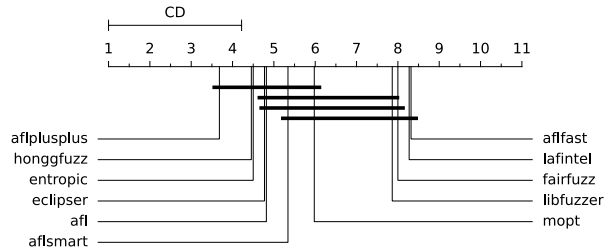

Figure 3: Critical difference diagram from Main Experiment.

on the statistical test results that the diagram depicts, there's no statistically significant difference between the seven highest ranking fuzzers. This is indicated by the bold line connecting these fuzzers. AFL++, however, is significantly better than the last four fuzzers, Honggfuzz is better than the last three, and so on.

RQ: Do different fuzzers cover different parts of the benchmarks?

FuzzBench keeps track of which code region was covered by each fuzzer. This allows users to answer questions such as whether anything is lost switching from one fuzzer to another, or whether one fuzzer complements another by covering code that the other does not. For each benchmark in the reports, FuzzBench provides a unique coverage plot (such as Figure 4) and a table of pairwise comparisons of the coverage covered by one fuzzer but not another (example provided in Figure 5). Of the 2,182,118 regions covered by any fuzzer, just 3,566 regions, or .163% were covered by only one fuzzer. No fuzzer found many regions that other fuzzers couldn't find.

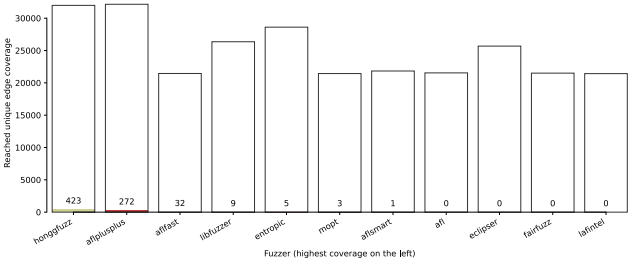

Figure 4: Unique coverage plot from freetype2-2017 in Main Experiment.

Fuzzers that did well in general (see Figure 3) tended to cover more unique regions. Table 2 shows the ranking of each fuzzer based on their unique regions covered per benchmark, as well as the total number of unique regions covered by each fuzzer (across all benchmarks). The average benchmark rank based on unique regions covered follows the same trend as the average benchmark rank based on median regions covered. These two rankings have a Pearson correlation of .660 with a p-value of .026.

Comparisons of different fuzzers also produced interesting findings. For example, Entropic, an academic improvement on libFuzzer, discovers almost a superset of the regions discovered by libFuzzer.

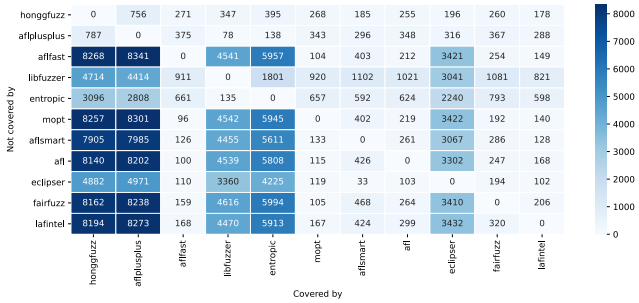

Figure 5: Pairwise coverage comparison plot from freetype2-2017 in Main Experiment.

Table 2: Fuzzer Rankings Based on Unique Regions.

Average Rank	Total Unique Regions
honggfuzz: 1.73	honggfuzz: 1121
aflplusplus: 2.14	entropic: 1119
entropic: 2.77	aflplusplus: 870
mopt: 3.41	fairfuzz: 113
eclipser: 3.45	libfuzzer: 69
fairfuzz: 3.5	eclipser: 65
libfuzzer: 3.59	mopt: 49
afl: 3.64	afl: 47
aflsmart: 3.73	aflfast: 46
lafintel: 3.77	lafintel: 38
aflfast: 3.82	aflsmart: 29

Based on the experiment results, we already know that Entropic is better than libFuzzer. But do we know if switching to Entropic from libFuzzer causes us to lose coverage? Initially on ClusterFuzz [5] — the system behind OSS-Fuzz — both Entropic and vanilla libFuzzer were used, with similar amount of resources provided for them. The reason for this approach was to increase diversity, even if it meant using fuzzer configurations considered less than optimal. Diversity might improve results compared to when only "optimal" configurations were used, as other configurations may be beneficial in particular scenarios. According to the coverage data from this experiment, however, Entropic covered more unique regions than libFuzzer on 18 benchmarks and covered 10,964 regions not covered by libFuzzer; libFuzzer covered more unique regions than Entropic on just one benchmark and covered only 446 regions not covered by Entropic. This implies the approach ClusterFuzz initially took for Entropic and libFuzzer was incorrect. There is almost no coverage missed when using Entropic instead of libFuzzer (446 regions is just over 1 region per trial run in this experiment). Entropic covered almost everything covered by libFuzzer and much more.

5.1 UNIFUZZ Comparison

It's worth comparing these results to those found by other papers that evaluate fuzzers. Table 3 compares the experiments used by papers introducing FuzzBench and other benchmark suites: UNIFUZZ and Magma. Li et al. used UNIFUZZ [32] to benchmark fuzzers in 2020. The following fuzzers are benchmarked by both FuzzBench and UNIFUZZ: AFL, AFLFast, Honggfuzz, and MOpt-AFL. The ranking of these fuzzers according to either metric offered by FuzzBench is Honggfuzz, AFL, MOpt-AFL, AFLFast. Li et al. evaluates fuzzer performance using number of unique bugs, bug severity, bug rareness, speed of finding bugs, and coverage. Table 3 compares the fuzzer rankings according to FuzzBench and UNIFUZZ. UNIFUZZ doesn't provide a ready-made aggregate result for number of unique bugs; instead, they do qualitative analysis of results on each benchmark. Their analysis is that (1) no fuzzer does better on every benchmark, (2) Honggfuzz, MOpt-AFL, AFL, and AFLFast perform best on 3, 3, 1, and 0 benchmarks, respectively. They also find that Honggfuzz, MOpt-AFL and AFLFast perform better than AFL on 11, 17, and 4 benchmarks, respectively. This seems to align with FuzzBench's ranking of Honggfuzz over AFL over AFLFast. However, UNIFUZZ's ranking of MOpt-AFL as best or second best contradicts FuzzBench's findings that it performs worse than AFL. This could be for a variety of reasons.

Table 3: Comparison of benchmark suites.

Suite	Trials	Duration	Benchmarks	Fuzzers
FuzzBench	20	23 hours	22	11
UNIFUZZ	30	24 hours	24	8
Magma	20	7 days	26	7

Table 4: Fuzzers ranked by FuzzBench and UNIFUZZ.

FuzzBench	UNIFUZZ (unique bugs)	UNIFUZZ (rare)	UNIFUZZ (rare bugs)
Honggfuzz	Honggfuzz- MOpt-AFL	MOpt-AFL	MOpt-AFL
AFL		Honggfuzz	Honggfuzz
MOpt-AFL	AFL	AFLFast	AFL
AFLFast	AFLFast	AFL	AFLFast

One likely reason for this different ranking is how FuzzBench invokes AFL-based fuzzers. As described in Section 7.1, FuzzBench uses AFL-based fuzzers with the `-d` option to skip deterministic steps. In early experiments, such as AFL Deterministic Experiment, FuzzBench did not use this option to invoke AFL. When this was changed in AFL non-Deterministic Experiment, AFL started doing better than MOpt-AFL. We think using `-d` is the right choice for benchmarking AFL, and we justify this more in Section 7.1.

5.2 Magma Comparison

We also compare our results with Magma's [30], reported in 2020. The following fuzzers are benchmarked by both FuzzBench and Magma: AFL, AFL++, AFLFast, FairFuzz, Honggfuzz, and MOpt-AFL. Magma uses unique bug discovery as the metric for evaluating fuzzer performance. They found that Honggfuzz and MOpt-AFL did far better than other fuzzers. Honggfuzz and MOpt-AFL did better on 4 and 3 benchmarks, respectively. They also found that AFL, AFLFast, and AFL++ "performed similarly against most targets". The findings of Hazimeh et al. regarding AFL++ and MOpt-AFL appear to contradict ours. We found that AFL++ does better than most fuzzers that we both evaluated, while MOpt-AFL does not

Table 5: Ranking by average score on reproduced experiments.

Main Experiment	Reproduce Experiment 1	Reproduce Experiment 2
Honggfuzz: 97.76	Honggfuzz: 97.05	AFL++: 97.35
AFL++: 96.53	AFL++: 96.47	Honggfuzz: 97.30
Eclipser: 95.79	Eclipser: 96.13	Eclipser: 96.02
Entropic: 95.47	Entropic: 95.51	Entropic: 95.63
libFuzzer: 91.10	libFuzzer: 90.88	libFuzzer: 91.26
AFLSmart: 90.73	AFLSmart: 90.76	lafintel: 90.85
lafintel: 90.59	lafintel: 90.51	AFL: 90.47
AFL: 90.59	AFL: 90.30	AFLSmart: 90.33
MOpt-AFL: 90.48	MOpt-AFL: 90.23	MOpt-AFL: 90.17
AFLFast: 88.25	AFLFast: 88.32	AFLFast: 88.41
FairFuzz: 85.86	FairFuzz: 86.24	FairFuzz: 85.70

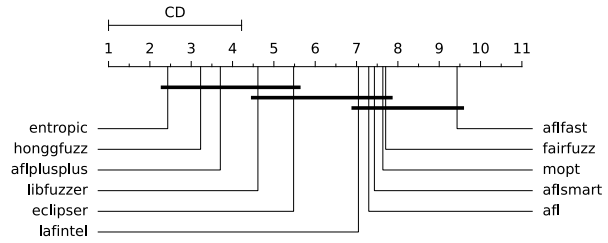

Figure 6: Critical difference diagram from No Seed Experiment.

do much better than the fuzzers we both evaluated. We found this with both code and bug coverage (see Bug Experiment). There are several potential reasons for these differences. First, we evaluated slightly different versions of these tools. Second, FuzzBench uses AFL-based fuzzers with the `-d` flag to skip deterministic steps. As we discussed earlier in this section and address more in Section 7, removing this flag can significantly improve performance of AFL-based fuzzers other than MOpt-AFL. This includes AFL++. Third, AFL++ did not use the cmplog feature in Hazimeh et al., yet it did use the feature in our evaluation. We found improved performance in AFL++ relative to MOpt-AFL when this feature was enabled (compare AFL++ non-Cmplog Experiment to AFL++ Cmplog Experiment).

6 EVALUATION OF METHODOLOGY AND EXPERIMENT PARAMETERS

This section evaluates whether FuzzBench experiments give sound and reproducible results. In FuzzBench, different parameters of an experiment (such as the number of benchmarks, number of trials, length of trials, use of seed inputs, dictionaries, etc.) can be easily configured. However, beyond flexibility, it is important to provide good defaults. In this section, we look at the impact of different parameter choices.

RQ: Are FuzzBench experiments reproducible?

First we look at whether running an experiment again gives us the same results. We ran two experiments, Reproduce Experiment 1 and Reproduce Experiment 2, with the same parameters used in the previous section (Main Experiment). Rankings based on average score (Table 5) and average rank (Table 6) are similar across all experiments. Note that the slight changes in order are expected, as they are due to the closeness of average score/rank values. Considering the clusters of statistically significantly different (or similar) fuzzers, reported in Figure 3, we get the same results every time. Therefore, we can conclude that the results are consistent and reproducible.

RQ: Is 23 hours enough?

The first experimental parameter we look at is the duration of the experiment. Klees et al. [31] suggest running trials for 24 hours. The FuzzBench service by default uses 23-hour trials so we can use Google Compute Engine's preemptible instances, which cannot run for longer than 24 hours, but are 5 times cheaper than standard instances. To validate our default 23-hour choice, we compared the result from our Main Experiment, which had 23-hour trials, with the results of an experiment with 7-day (168-hour) trials: 7 Day Experiment. As can be seen in Table 6 the average ranks are mostly similar. In fact, average rank in Main Experiment and average rank in 7 Day Experiment has a Pearson correlation of .927, meaning they are strongly correlated. However, there are some differences in ranking and some greater differences on individual benchmarks. Because of the strong correlation between the two rankings, we think this confirms prior work [31] and mostly validates that 23-hour trials strikes a good balance between result accuracy and the price of the experiments. However, more research is needed here.

RQ: Should example inputs be used as seed corpus?

Another experimental parameter we evaluate is the impact of the seed corpus. In the main experiment we use a mix of benchmarks with and without seed corpora. This makes sense because in real-world settings, such as on OSS-Fuzz, some fuzz targets do not come with a seed corpus, while some do. Previous research has found that observed fuzzer performance can vary significantly based on the seed corpus [31]. To see how this impacts our evaluation of fuzzers, we did an experiment where no seed corpora were used, even for benchmarks that provide them: No Seed Experiment. As can be seen in Table 6, Honggfuzz, Entropic, and AFL++ still perform at the top in this experiment. However libFuzzer does much better in this experiment. In fact, it ranks higher than Eclipser, which is the fifth ranked fuzzer in this experiment. Every one of the fuzzers that rely on AFL's traditional instrumentation (AFL, MOpt-AFL, FairFuzz, AFLFast, lafintel, Eclipser, and AFLSmart) ranks below the fuzzers that use different instrumentation (Entropic, Honggfuzz, AFL++, libFuzzer). As Figure 6 shows, Entropic, Honggfuzz and AFL++ perform statistically significantly better than all of the AFL-based fuzzers (including AFL) except Eclipser.

To understand the underlying reason for the change in ranking, we examined results at the benchmarks level. In the benchmarks: bloaty_fuzz_target, freetype2-2017, libpcap_fuzz_both, proj4-2017-08-14, systemd_fuzz-link-parser, and vorbis-2017-12-11, the fuzzers: MOpt-AFL, FairFuzz, AFLSmart, AFLFast, and AFL make little progress in covering code throughout the experiment. In some of these benchmarks, Eclipser and lafintel can make progress, but in others, they cannot. A similar pattern can be seen in the benchmark libpcap_fuzz_both (which has no seeds) in Main Experiment. We believe the cause of this pattern is likely the same. All fuzzers that can make progress in the above benchmarks add special instrumentation for comparisons, with the exception of Eclipser. AFL++, libFuzzer, Entropic, and Honggfuzz instrument functions like `strcmp`

Table 6: Average rank from various experiments.

Main Experiment	7 Day Experiment	No Seed Experiment	OSS-Fuzz Corpus Experiment	No Dictionary Experiment	Reproduce Experiment 1	Reproduce Experiment 2
AFL++: 3.68	Entropic: 3.84	Entropic: 2.43	AFL++: 4.10	AFL++: 3.18	AFL++: 3.55	AFL++: 3.43
Honggfuzz: 4.45	AFL++: 3.93	Honggfuzz: 3.23	AFL: 4.78	Honggfuzz: 4.07	Eclipser: 4.25	Eclipser: 4.30
Entropic: 4.50	Eclipser: 4.09	AFL++: 3.70	AFLSmart: 4.88	Entropic: 4.64	Entropic: 4.41	Honggfuzz: 4.50
Eclipser: 4.77	AFLSmart: 4.80	libFuzzer: 4.61	Eclipser: 4.88	Eclipser: 4.70	Honggfuzz: 4.82	Entropic: 4.73
AFL: 4.82	Honggfuzz: 5.05	Eclipser: 5.48	Entropic: 5.60	AFL: 5.16	AFLSmart: 5.07	AFL: 5.09
AFLSmart: 5.34	AFL: 5.34	lafintel: 7.05	MOpt-AFL: 5.72	AFLSmart: 5.41	AFL: 5.32	AFLSmart: 5.27
MOpt-AFL: 5.98	MOpt-AFL: 6.32	AFL: 7.30	Honggfuzz: 5.98	MOpt-AFL: 6.59	MOpt-AFL: 6.73	MOpt-AFL: 6.64
libFuzzer: 7.86	lafintel: 6.91	AFLSmart: 7.43	AFLFast: 6.70	libFuzzer: 7.30	FairFuzz: 7.68	libFuzzer: 7.61
FairFuzz: 8.00	libFuzzer: 7.89	MOpt-AFL: 7.64	libFuzzer: 7.00	FairFuzz: 8.11	libFuzzer: 7.68	FairFuzz: 7.77
lafintel: 8.27	AFLFast: 8.80	FairFuzz: 7.70	lafintel: 7.68	lafintel: 8.39	lafintel: 8.11	AFLFast: 8.32
AFLFast: 8.32	FairFuzz: 9.05	AFLFast: 9.43	FairFuzz: 8.70	AFLFast: 8.45	AFLFast: 8.39	lafintel: 8.34

and add compared string constants to a dictionary. Lafintel instruments comparisons by breaking down multi-byte comparisons into multiple basic blocks so the fuzzer can tell when it gets further into a comparison than before. Eclipser does not instrument comparisons but tries to solve for them using symbolic execution, and its ability to outperform fuzzers which do not instrument comparisons has been noted by its authors [12]. The fuzzers that do not make progress here do not instrument comparison functions. Results from other experiments confirm that comparison instrumentation is responsible for progress in libpcap_fuzz_both. In AFL++ non-Cmplog Experiment and every experiment before it, AFL++ made no progress on libpcap_fuzz_both. But, in AFL++ Cmplog Experiment it made progress on covering the benchmark. What changed is that AFL++'s "cmplog" feature, which instruments comparison functions, was enabled. FuzzBench highlights effective techniques by allowing these types of comparisons across a large number of fuzzers.

In summary, we can confirm that results will be different with and without seeds. Evaluating without seeds can be useful, as it brings out the differences between fuzzers that can or cannot break through some comparisons/branches. On the other hand, fuzzer users who typically use seed inputs might be more interested in results from Main Experiment, since it used seeds.

RQ: Should an existing, well-established, saturated corpus be used as seed corpus?

After realizing that the starting corpus significantly affects results, we wanted to perform another experiment that reflects a common real-world scenario. In the real world, it is actually rare that a fuzzer is started from no corpus. Most of the time when fuzzing a target in OSS-Fuzz, a fuzzer is starting from a corpus that was created by many CPU years of fuzzing effort by AFL(++), libFuzzer and Honggfuzz.

We conducted an experiment OSS-Fuzz Corpora Experiment using corpora from OSS-Fuzz to explore this scenario. As one might expect, most fuzzers performed similarly to one another, and made only minimal progress relative to the well-established existing corpus. The worst performing fuzzer in this experiment on average found 97.11% of the regions found by the best performing fuzzer on each benchmark. In contrast, in Main Experiment the worst performing fuzzer found 85.86% of the regions found by the best fuzzer on each benchmark. On most benchmarks, fuzzers performed virtually identically. No fuzzer appears a lot better at finding more coverage than others when starting from OSS-Fuzz corpora. AFLSmart ranks noticeably better than it did in Main Experiment. However the difference is non-significant, and is likely due to random chance. Another reason to be skeptical of this difference is bias in the experiment. Because the seed corpora were generated by spending multiple CPU-centuries running AFL, AFL++ (which OSS-Fuzz switched to in January 2021), libFuzzer, and Honggfuzz, the experiment is biased against them and favors fuzzers such as AFLSmart which are not run on OSS-Fuzz. In summary, we do not think a saturated corpus should be used for a first evaluation, because it is very hard for fuzzers to augment it. However, if a fuzzer is found to perform well, that result is quite useful, since real-word fuzzing (e.g., OSS-Fuzz) does run fuzzers with saturated corpora.

RQ: Should dictionaries be used?

Dictionaries are a feature supported by most coverage guided fuzzers. They allow someone using a fuzzer to provide the fuzzer with tokens to insert during fuzzing. Like seed corpora, these are optional for fuzzing and in OSS-Fuzz, there is a mix of fuzz targets with and without dictionaries. Thus, FuzzBench has a mix of benchmarks with and without dictionaries. To explore how dictionaries affect our results, we ran an experiment (No Dictionary Experiment) without dictionaries. As can be seen in Table 6, the rankings of fuzzers relative to one another is only slightly impacted by removing this parameter. In summary, the choice of whether to enable dictionaries does not have a significant impact on our overall results. Note that researchers are free to adjust all of the parameters discussed above, if needed over the course of their research. But overall, based on these experiments, FuzzBench's default configuration appears to provide a reliable standard for most evaluations.

7 CASE STUDIES OF TOOL DEVELOPMENT WITH FUZZBENCH

This section discusses specific case studies where FuzzBench was used to improve fuzzers.

7.1 AFL Configuration Insights

AFL's default behavior is to start with "deterministic fuzzing steps", which includes mutations such as flipping bits to trigger integer arithmetic errors. These steps can be skipped with the `-d` flag. The downside is that the deterministic stage "can take several days" [46] and may not show coverage improvements right away. The upside is that with this stage, coverage overall is supposed to better even if it is initially less broad. After this step is complete, AFL moves on to its ordinary mode of fuzzing that randomly mutates inputs (splicing, bitflips, insertions, etc.). Most evaluations of fuzzers [31] (including most in this paper) evaluate them for only 24 hours or less. This violates an underlying assumption of deterministic

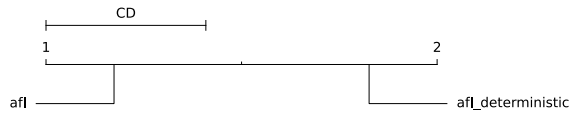

Figure 7: Critical difference diagram from experiment with AFL with -d (afl) and AFL without -d (afl_deterministic).

steps: that a lot of time can be spent up front so that fuzzing is more efficient later on. Another reason to skip deterministic steps is because it isn't used very often when fuzzing in the real world. For example, in OSS-Fuzz, a heuristic was used so that fuzzing with `-d` was done only once, right after the fuzzer was added to OSS-Fuzz. Similarly, when using AFL in parallel, only the primary instance runs the deterministic stage, while secondary instances do not. These facts raise the question of whether it makes more sense to benchmark AFL with `-d` or not. In many ways, using `-d` is more likely to represent real-world settings where AFL is used. We ran an experiment (AFL non-Deterministic Experiment) to investigate differences in performance between AFL with and without using the deterministic steps (by using the `-d` command line option). As can be seen in Figure 7, skipping the deterministic stage caused AFL to perform statistically significantly better than AFL with the deterministic stage (afl_deterministic). This suggests that if runs are short (e.g., 24 hours), deterministic steps should be disabled. Thus, in the experiments above FuzzBench uses the `-d` option when invoking AFL-based fuzzers. However, this is not what most other research papers do. For example, we found that MOpt-AFL does no better than AFL without deterministic steps even though it performs statistically significantly better than AFL with deterministic steps. This may be explained by the fact that MOpt-AFL implements a "pacemaker fuzzing mode which selectively avoids the time-consuming deterministic stage" [33]. Based on FuzzBench's findings on deterministic steps, AFL++ skips them by default, which improves fuzzing in many real-world scenarios.

7.2 Libfuzzer Improvements

LLVM's libFuzzer is one of the most widely-used fuzzing engines. It is run on tens of thousands of cores continuously by ClusterFuzz and OSS-Fuzz [5]. Because of libFuzzer's widespread use, improvements to libFuzzer have a large positive impact on software security, while regressions have a large negative impact. FuzzBench has detected weaknesses in libFuzzer, driven improvements, and verified the lack of regressions. Furthermore, FuzzBench has improved fuzzing effectiveness by non-expert users, since making the default settings better improves fuzzing for users that don't change them. We highlight select examples of these improvements here.

A first example is related to libFuzzer's CrossOver mutation, which mixes two test cases. When libFuzzer developers noticed libFuzzer performed worse on FuzzBench's sqlite3_ossfuzz benchmark they investigated and determined that libFuzzer was producing considerably larger test cases than AFL. They determined that this was caused by a bug in the CrossOver implementation, which they fixed [17].

A second example is related to seed pruning, the process of deleting seeds which produce the same coverage as other, smaller seeds, allowing fuzzers to ignore what were previously considered useless inputs and reducing storage space. FuzzBench demonstrated that initial seeds, which in many cases are handpicked by users because they are interesting, can still be useful in helping drive new coverage. Thus, an option for keeping these seeds was changed to reflect this finding [15].

A third example is related to the run time replacement of memory-related functions. This is common practice in the fuzzing community, with the compiler injecting calls to sanitizer runtimes when invoking functions like `memcmp` and `strcmp`. Previously, libFuzzer required the AddressSanitizer be linked in to provide these functions, but FuzzBench identified the need for libFuzzer to have a dedicated runtime when no LLVM sanitizer was being used [18]. FuzzBench results showed an issue with coverage, leading to the discovery and remediation of this flaw.

A final example highlights how FuzzBench can be used to drive optimal defaults, improving coverage for non-experts who may not know how to tune the behavior of libFuzzer. FuzzBench showed that the entropic seed-scheduler in libFuzzer outperforms the vanilla scheduler [9]; thus, the entropic seed-scheduler was made the default [16].

7.3 AFL++ Improvements

AFL++ is another popular production-ready fuzzer (1,604 stars on Github). The authors of AFL++ not only used FuzzBench to evaluate their tool in their published paper [22], they have also been using FuzzBench extensively to evaluate features and improvements and thus drive the tool's development. Part of AFL++'s success is due to complex integrations of code bases and fuzzing concepts from the literature. One such integration ported the concept of "power schedules" from AFLFast [10] by integrating parts of the AFLFast code base, which inadvertently introduced a performance regression when using non-AFLFast enhanced runs. FuzzBench helped identify the discrepancy between regular AFL and AFL++ after this code change, indicating a regression. Without this insight, this subtle bug would have gone unnoticed. Having adopted power schedules in AFL++, Boehme et al. [10] focused their efforts on power schedule improvements on the AFL++ version and used FuzzBench to verify the improvements.

FuzzBench is also driving better defaults, helping to reduce the barrier to entry and improve results for those who adopt AFL++. As mentioned earlier in this section, data from FuzzBench led to AFL++ disabling deterministic steps by default in AFL++ 3.0.

AFL++ also developed and tested an improved seed selection schedule in version 3.0 which was benchmarked against vanilla AFL over many FuzzBench runs. For example, when the discovery of new coverage-increasing inputs decreases, more splicing mutations can improve coverage. Additionally, it is better not to trim newly found inputs. It can be hard to quantify these improvements and understand the impact against different types of software without a comprehensive test suite.

FuzzBench also helped AFL++ researchers avoid dead ends by providing an easy way to test proposed improvements. For example, weight havoc mutations in AFL++ appeared to have little

benefit in practice. This early identification of limitations has enabled the research community to focus on fuzzing mechanisms that can meaningfully improve fuzzing effectiveness.

7.4 Honggfuzz Improvements

The author of Honggfuzz published a series of these improvements on the groups.google.com/g/fuzzing-discuss mailing list. These improvements included things such as adding strings/memory passed to comparison functions to the dynamic dictionary, or adding a mutation to change ASCII integers. An interesting thing about some of these improvements is that many of these features were initially implemented by libFuzzer. Thanks to FuzzBench, the developer of Honggfuzz was able to find features from other fuzzers, prove they were useful, and then add the features to Honggfuzz. Not only does this have the benefit of improving Honggfuzz, it also demonstrates the validity of techniques used by libFuzzer and thus benefits libFuzzer as well.

8 RELATED WORK

Klees et al. [31] studied the evaluations of recent academic fuzzing papers. They found a lack of scientific rigor which made results unreproducible, leading to "wrong or misleading assessments". As a result of their work, they released guidelines for the fuzzing community. FuzzBench follows these guidelines to provide statistically sound, reproducible results: run multiple trials, deduplicate bugs, supports flexible parameter selections, and provide a diverse set of real-world programs.

FuzzBench is not the first attempt to create a benchmarking platform for fuzzers. The Competition on Software Testing [8] is an annual competition to test software techniques. The benchmarks are small (a few 100 LoC) and are intended to test the ability of tools to adapt to specific code patterns (e.g., arrays, loops, etc.). Their work and ours are complementary, as it is useful to consider both micro and macro behavior of tools to understand their characteristics. Another benchmark suite is Google Fuzzer Test Suite [25], which was an early prototype by Google. It was primarily a carefully selected set of real-world software for fuzzer evaluation. However, it did not have the advanced features of the FuzzBench platform and service that reduce the cost of integration and evaluation.

There are also several benchmark suites that use artificially introduced bugs. LAVA [20] is an automated bug injection technique, and also set of benchmark programs containing synthetic bugs injected with the technique. Rode0day [21] is another service that uses the LAVA technology to create artificially buggy binaries. The DARPA CGC [13] benchmark suite also uses artificial vulnerabilities. All of these benchmarks, however, are much smaller than the real-world programs used in FuzzBench. Further, it has been shown that generalizing from synthetic bug benchmarks to real-world programs seems to be hard. Geng et al. [24] and Bundt et al. [11] studied artificially injected bug benchmarks. They found that the bugs "significantly differ from real bugs", and that fuzzers performing well on synthetic benchmarks do not "find any organic bugs". This implies fuzzing techniques benchmarked with these datasets may not generalize to real-world software. To mitigate these problems, FuzzBench uses real-word benchmarks containing real bugs from OSS-Fuzz, the largest open-source fuzzing initiative.

Magma [30] is a collection of widely used open-source programs with a long history of security-critical bugs. Like FuzzBench, the authors target real-world software but they focus on bug finding as the only metric. FuzzBench provides both code coverage and bug finding as metrics out of the box. This is important because bugs are sparse, so a fuzzer's ability to cover more of the code where bugs exist may go unnoticed [32]. UNIFUZZ [32] is a recent benchmarking work that uses real-world software, considers both bugs and coverage as metrics and suggests additional metrics such as the speed to find bugs. Although FuzzBench does not provide all the metrics used by UNIFUZZ by default, there is support for them and we have enabled them for certain academic groups already. A major difference between FuzzBench and UNIFUZZ, and the previously mentioned benchmark suites, is that FuzzBench was designed primarily with users in mind: the focus is on running experiments as easily as possible.

To the best of our knowledge, FuzzBench is the only fuzzer benchmarking service. Due to the scalability of the FuzzBench service, it can provide results within a day, and thus reduce the barrier to entry for researchers and help them quickly iterate over ideas. Since we released FuzzBench in 2020 [35], it has been actively used at Google and by the broader fuzzing community both in industry and academia [9, 36].

9 CONCLUSION

We created FuzzBench with the goal of facilitating open science, rigorous evaluation, and reproducibility. The service not only makes unbiased, large-scale evaluations easy, it also makes these evaluations free for researchers. Since its release in the past year, FuzzBench has already been used by dozens of researchers from academia and industry to run more than 150 experiments. Further, FuzzBench is a fully open, community driven platform. It has also already received many great contributions, and we continue to encourage the community to continue to improve the platform and the techniques used to evaluate testing tools. With the success stories described in this paper behind it, FuzzBench is in our opinion poised to become the gold standard for fuzzer evaluation. We hope that better evaluations leads to better science, more adoption, and greater impact for fuzzing research and tools.

ACKNOWLEDGMENTS

FuzzBench would not be possible without the help of many contributors. We would like to thank coworkers who implemented features and contributed ideas: Oliver Chang, Franjo Ivančić, Kostya Serebryany, Matt Morehouse, Martin Barbella, Ammar Askar, Zhicheng Cai, Dokyung Song, Ruikang Shi, and Tanishq Rupal. We would also like to thank community members who have contributed valuable ideas and features, particularly: Marc Heuse, Marcel Böhme, and Josh Bundt. Thank you to Kara Olive who provided important editing of this paper. Lastly, we would like to thank our reviewers for their feedback.

REFERENCES

[1] 2021. OSS-Fuzz Bugs. https://bugs.chromium.org/p/oss-fuzz/issues/list?q=-status%3Aduplicate%2Cwontfix%20label%3Aclusterfuzz%20opened%3C2021-04-06&can=1

[2] ACM SIGPLAN. 2018. Empirical Evaluation Guidelines. http://www.sigplan.org/Resources/EmpiricalEvaluation/

[3] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google Testing Blog. (Dec. 2016). https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[4] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. 2011. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering* 38, 2 (2011), 258–277.

[5] Abhishek Arya and Chang Oliver. 2019. ClusterFuzz: ClusterFuzz. https://www.blackhat.com/eu-19/briefings/schedule/index.html#clusterfuzz-fuzzing-at-google-scale-17505 Black Hat Europe.

[6] 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.

[7] FuzzBench Authors. 2021. FuzzBench. https://github.com/google/fuzzbench/tree/d51da57c211af226dd7854085d9fc80070205738/fuzzers/symcc_afl.

[8] Dirk Beyer. 2020. Competition on Software Testing. https://test-comp.sosy-lab.org/2020/

[9] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. 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 (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/3368089.3409748

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[11] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. *Hong Kong* (2021), 14.

[12] 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*. 736–747.

[13] DARPA. 2016. DARPA Cyber Grand Challenge Final Event Archive. http://www.lungetech.com/cgc-corpus/

[14] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.

[15] Song Dokyung. 2020. Add an option to keep initial seed inputs around. https://reviews.llvm.org/D86577

[16] Song Dokyung. 2020. Enable entropic by default. https://reviews.llvm.org/D87476

[17] Song Dokyung. 2020. Fix arguments of InsertPartOf/CopyPartOf calls in CrossOver mutator. https://github.com/llvm/llvm-project/commit/c10e63677f5d20f18010f8f68c631ddc97546f7d

[18] Song Dokyung. 2020. Link libFuzzer's own interceptors when other compiler runtimes are not linked. https://reviews.llvm.org/D83494

[19] Song Dokyung. 2020. Scale energy assigned to each input based on input execution time. https://github.com/llvm/llvm-project/commit/5cda4dc7b4d28fcd11307d4234c513ff779a1c6f

[20] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

[21] Andrew Fasano. 2019. Rode0day: A Year of Bug-Finding Evaluations.

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi

[23] FuzzBench. 2021. The report – FuzzBench. https://google.github.io/fuzzbench/reference/report/

[24] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. 2020. An Empirical Study on Benchmarks of Artificial Software Vulnerabilities. (March 2020). https://arxiv.org/abs/2003.09561v1

[25] Google. 2017. Fuzzer Test Suite. Google. https://github.com/google/fuzzer-test-suite

[26] Google. 2021. Automate your workflow from idea to production. https://github.com/google/fuzzing/actions

[27] Google. 2021. Coverage Guided Fuzz Testing. https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/

[28] Google. 2021. fuzzing - Google Scholar. https://scholar.google.com/scholar?q=fuzzing&hl=en&as_sdt=0%2C5&as_ylo=2014&as_yhi=

[29] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid : Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference*. ACM, Austin USA, 360–372. https://doi.org/10.1145/3427228.3427266

[30] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (Nov. 2020), 49:1–49:29. https://doi.org/10.1145/3428334

[31] 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*. ACM, Toronto Canada, 2123–2138. https://doi.org/10.1145/3243734.3243804

[32] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2020. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. *arXiv:2010.01785 [cs]* (Oct. 2020). arXiv:2010.01785 [cs] http://arxiv.org/abs/2010.01785

[33] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.

[34] Jonathan Metzman and Asra Ali. 2021. Developers are Buzzing on Fuzzing. https://thenewstack.io/developers-are-buzzing-on-fuzzing/

[35] Jonathan Metzman, Abhishek Arya, and László Szekeres. 2020. FuzzBench: Fuzzer Benchmarking as a Service. https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html

[36] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-Based Symbolic Execution for Binaries. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, Virtual. https://doi.org/10.14722/ndss.2021.24118

[37] Kostya Serebryany. 2015. libFuzzer – a Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html

[38] Kostya Serebryany. 2021. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, USA, 28.

[40] Paras Shah. 2021. Continuous Fuzzing with Go at Dgraph. https://dgraph.io/blog/post/continuous-fuzzing-with-go/

[41] L. Simon and A. Verma. 2020. Improving Fuzzing through Controlled Compilation. In *2020 IEEE European Symposium on Security and Privacy (EuroS P)*. 34–52. https://doi.org/10.1109/EuroSP48549.2020.00011

[42] Adith Sudhakar, Mohit Arora, and Souheil Moghnie. 2021. Focus on Fuzzing: Fuzzing Within the SDLC. https://safecode.org/focus-on-fuzzing-fuzzing-within-the-sdlc/

[43] Robert Swiecki. 2015. Honggfuzz. https://honggfuzz.dev/

[44] Robert Swiecki. 2020. Improving honggfuzz for fuzzbench part 3. https://groups.google.com/g/fuzzing-discuss/c/rv59P0svXjI

[45] Michał Zalewski. 2014. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/

[46] Michał Zalewski. 2014. Fuzzing with Afl-Fuzz — AFL 2.53b Documentation. https://afl-1.readthedocs.io/en/latest/fuzzing.html

[47] Andreas Zeller. 2019. Andreas Zeller's Old Blog: When Results Are All That Matters: Consequences. https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters.html

[48] Andreas Zeller. 2019. Andreas Zeller's Old Blog: When Results Are All That Matters: The Case of the Angora Fuzzer. https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters-case.html