Comparing Fuzzers on a Level Playing Field with FuzzBench

Dario Asprone*, Jonathan Metzman†, Abhishek Arya†, Giovani Guizzo*, Federica Sarro*
*University College London, London
† Google

***Abstract*—Fuzzing is a testing approach commonly used in industry to discover bugs in a given software under test (SUT). It consists of running a SUT iteratively with randomly generated (or mutated) inputs, in order to find as many as possible inputs that make the SUT crash. Many fuzzers have been proposed to date, however no consensus has been reached on how to properly evaluate and compare fuzzers. In this work we evaluate and compare nine prominent fuzzers by carrying out a thorough empirical study based on an open-source framework developed by Google, namely FuzzBench, and a manually curated benchmark suite of 12 real-world software systems. The results show that honggfuzz and AFL++ are, in that order, the best choices in terms of general purpose fuzzing effectiveness. The results also show that none of the fuzzers outperforms the others in terms of efficiency across all considered metrics, that no particular bug affinity is found for any fuzzer, and that the correlation found between coverage and number of bugs depends more on the SUT rather than on the fuzzer used.**

***Index Terms*—Fuzzing, Software Testing, FuzzBench, Empirical Study**

I. Introduction

Fuzzers are commonly used in industry during the software testing process to find inputs that, when fed to the software under test (SUT), will make it crash [4], [27]. There is also a great deal of research on producing new fuzzers which utilize innovative techniques in an effort to improve their bug-finding ability [15], [26]. In spite of this activity, there is no unified methodology or benchmark suite for fuzzers. Thus, it has been challenging to carry out a comprehensive benchmark analysis to compare fuzzers, and assess if one performs better than others. This has led to different studies using different evaluation techniques including different target programs, different environments and/or different parameters [15], which impacts a user's ability to reliably choose the most suitable fuzzer to their purposes. As shown by Klees et al. [15], there is no established consensus on how to evaluate fuzzers or compare them against each other, as evidenced by the fact that each of the 32 papers they analysed used a different evaluation methodology. An empirical study that compares the currently most prominent fuzzers against each other, based on a sound methodology, including taking into account the guidelines by Klees et al., metrics based on real-world bugs, and a diverse enough set of benchmarks seems overdue. Our work sought to fill this gap.

In this paper, we aim to evaluate a set of relevant and widely used fuzzers in a robust and fair way, and to compare them against each other based on metrics that accurately reflect the fact that a fuzzer's main goal is to discover bugs in the SUT. To this end, we evaluate fuzzers' effectiveness and efficiency by assessing their ability to find crashes that match with a ground truth, a set of real-world bugs found by OSS-Fuzz; we also evaluate fuzzers on their ability to find additional crashes that are not present in our ground truth. In summary, we set to address two common issues in fuzzing evaluation: Having a weak or lacking methodology, and relying on proxy evaluation metrics that do not reflect real-world purposes.

To run our experiments in a systematic manner and with minimal variations between runs, we use FuzzBench [21], a framework designed by Google to facilitate fuzzer benchmarking. While initially FuzzBench only focused on comparing the code coverage of different fuzzers and ranking them according to this metric, we have extended it to rank fuzzers based on found bugs. Our work includes the following: extending FuzzBench in order to detect/record the crashes; implementing a mechanism to get a unique hash for each detected crash (for de-duplication); choosing a set of benchmarks that fit our requirements; devising a robust methodology to assess and rank fuzzers, and finally, running the chosen fuzzers on the selected benchmarks, while keeping track of which of the known and how many unknown bugs have been found by each fuzzer.

We compare nine prominent fuzzers (`AFL`, `AFLFast`, `AFLSmart`, `AFL++`, `mOpt`, `entropic`, `honggfuzz`, `libfuzzer`, `FairFuzz`) on their ability to find real-world bugs, or more specifically to trigger crashes related to bugs in the ground truth we have compiled from the OSS-Fuzz platform, as identified by the de-duplication functionality of ClusterFuzz. We also record and compare additional information, such as the achieved coverage and the number of crashes whose hashes do not match any bug in the ground truth (and thus can be considered potentially newly discovered bugs). This information, together the number of found ground truth bugs, is then analysed in three steps: (1) a report for each fuzzer is generated containing all the data gathered during our experiment; (2) the fuzzers are compared against one another based on various metrics and by using statistical significance and effect size analyses in order to establish whether we have found any relevant difference; (3) a selection of bugs found by each fuzzer is qualitatively analysed to understand if there is any correlation between fuzzers and the characteristics of the bugs they find.

Our results show that, across all metrics, `honggfuzz` and `AFL++` equally outperform all other fuzzers according to the Friedman test, with `AFLFast`, `AFL`, `entropic`, and `mOpt` following in a tight group. Two runner-up groups are formed, almost coinciding with the fuzzer families. When considering other aspects, such as effect size, number of unique bugs found, or total number of bugs found over all trials, `honggfuzz` performs better than all other fuzzers, followed closely by `AFL++`. Additionally, when considering efficiency, we find that the roles reverse: `honggfuzz` ends up at the bottom of the rankings, with `AFL++` performing slightly better, but still ending up towards the bottom based on two out of three metrics. `Fairfuzz` instead, which performed the worst in terms of efficacy, proved the most efficient fuzzer in finding both additional crashes and in expected time to find the last distinct crash.

In summary, the main contributions of our work are: (1) a large-scale rigorous empirical study comparing widely used, publicly available, fuzzers based on a solid methodology with real-world bugs; (2) a manually curated ground truth suite of real bugs from open-source projects; (3) open-source code and data available in FuzzBench [2] and our online appendix [3] to allow for reproduction, replication and extension of our work.

II. Background

This section provides some background about fuzz testing, the FuzzBench tool, and other tools used in our work.

A. Fuzz Testing

In automated software testing, fuzz testing (or *fuzzing*) is a relatively simple technique to discover bugs in a software. It consists in running the software under test iteratively with randomly generated or mutated inputs, aiming at finding those inputs that make the software crash. While straightforward in principle, coverage-guided fuzzing generally focuses on quantity over quality, generating numerous inputs in the hope that some of them will hit a bug in the code. In practice fuzzing has been shown to work better than other testing techniques, such as symbolic execution, and to find a larger number of bugs on the SUTs [31]. Fuzzers have found thousands of bugs in real-world software systems in recent years [23], [24], [34]. Next we explain some key terms commonly used for fuzzing.

De-duplication is a process through which crashes identified in a fuzzing campaign are tentatively clustered together according to the bug that is thought to have produced them. More specifically, the aim of the de-duplication process is to determine with a certain degree of accuracy whether two crashes are due to the same bug or have different causes. This is needed since the objective of fuzzing is not to find crashes, but to find the bugs underlying the crashes. It is worth noting that de-duplication is useful for two relevant groups, fuzzer users and fuzzer researchers and developers. De-duplication is useful for fuzzer users because it makes it easier to infer the underlying bug. De-duplication is useful for fuzzer researchers and developers because it allows for evaluation of fuzzers on the metric that most closely resembles the ultimate goal of fuzzers: finding bugs.

Previous work has shown that current de-duplication methods produce 1-2 orders of magnitude more "unique" crashes than there should be when analysed [15]. In our work we use ClusterFuzz's de-duplication technique [20], [23]. This technique reduces the number of crashes falsely believed to be unique, enough so that OSS-Fuzz automatically reports bugs to developers. As with any de-duplication technique, there is also the possibility that two different bugs are considered to be the same bug. In theory, it would be possible to use older target versions with known bugs, tracing back each crash to the bug causing it and then using the number of bugs discovered as a metric, but this would require having a dataset of programs with known, reproducible, and discoverable bugs by fuzzers.

The *seeds* are input files given by the user to the fuzzer, in order to jump-start the fuzzing campaign. Mutation based fuzzers start their mutation process on seeds, and as such seeds are vital in order to produce good results during fuzzing: using an empty file, a correctly formatted, or an invalid one can drastically affect a fuzzer's performance [15].

Sanitizers are compile time instrumentation for programs that are inserted by the compiler into executable code to check for specific bugs such as memory unsafety or undefined behaviour. This includes bugs such as buffer overflows and integer overflows. The type of errors detected depend on the specific sanitizer(s) used during compilation.

Oracles in fuzzing, as exemplified by their name, establish whether the behaviour of the SUT was correct or not. In a basic example, an oracle could be a script that, after running the SUT, reports no crash if the recorded exit code was 0, and reports a crash if the exit code was not 0. With this oracle a fuzzer can understand whether or not the input used triggered bugs such as memory corruption in the SUT, since bugs that violate memory safety can cause segmentation faults, which cause the program to return a nonzero exit code. This oracle is oftentimes used in fuzzing. However, to identify more bugs, the SUT can be compiled with sanitizers that crash the SUT during execution when a bug occurs, even if under normal circumstances the SUT would not crash. More complex oracles can be based on the execution reaching a buggy state specific to the SUT, or on whether the SUT passes taint analysis, but such oracles usually require modifying the SUT's source code.

B. FuzzBench, OSS-Fuzz and ClusterFuzz

FuzzBench [21] is an open-source tool developed by Google to provide "fuzzer benchmarking as a service" [21]. Benchmarks in FuzzBench are based on the fuzz targets in OSS-Fuzz, which is a continuous fuzzing service for open-source projects [23]. FuzzBench is based on docker, and allows for local execution of experiments and for integration of new fuzzers and new benchmark programs to the already existing library. FuzzBench can also be modified to customise the reports it produces in order to include additional data or perform different analyses. In this work we use the FuzzBench platform to systematically run a series of fuzzing campaigns with 9 fuzzers over 12 different open-source software programs, while keeping the testing environment consistent between runs.

OSS-Fuzz [8], [23] is a continuous fuzzing service run by Google that uses `AFL++` (and previously `AFL`), `libfuzzer`, and `honggfuzz` to fuzz the latest version of an open-source project, in order to find bugs, and report them to the developers who then fix the bugs. The software that is fuzzed by OSS-Fuzz is "user" supplied, i.e., the users are the developers of the projects in question. At the time of writing, OSS-Fuzz [23] has reported more than $35,000$ bugs, in over 400 open source projects. The same considerations made for all the other candidate datasets still apply: not all bugs are confirmed, some are duplicates or "won't fix" bugs, some others are too new or too old and they do not have the proper data recorded, plus each bug only appears in a specific range of program versions (we can only fuzz one), so we would need to find the buggiest version of each program through some analysis.

ClusterFuzz [20] is continuous fuzzing infrastructure developed by Google that runs fuzzers such as `AFL++`, `libfuzzer`, and `honggfuzz` during the development process. ClusterFuzz is the back-end for OSS-Fuzz, OSS-Fuzz uses ClusterFuzz for fuzzing and bug reporting. ClusterFuzz can be used as a library allowing users to use some of its functionality without running the entire infrastructure. This includes the crash hashing functionality we used in this work.

III. Empirical Study Design

The main goals of our work are to compare a set of prominent fuzzers in a comprehensive, rigorous and fair way, to provide a unified dataset of real-world software and bugs, and to address, at least partly, the issues found in many recent work that proposes and evaluates fuzzers [15].

To this end, we evaluate the fuzzers based on three main criteria, which correspond to our first three research questions: effectiveness, efficiency, and bug type affinity. We also investigate the relationship between coverage and the number of crashes or bugs found by a certain fuzzer in a given SUT , which is our fourth and last research question. We discuss each research question in detail in Section III-A.

As a benchmark to compare the fuzzers, we identified a set of 12 real-word software systems containing previously bugs found by fuzzers from OSS-Fuzz, which are further described in Section III-B.

After integrating this set of software projects with FuzzBench, we run them within a similar environment used by OSS-Fuzz, in order to ascertain how many of the bugs originally reported in OSS-Fuzz could be reproduced in FuzzBench. As expected we were unable to trigger some of the bugs, even though both environments are similar. From here on, we refer to the number of bugs in the ground truth for each benchmark as the number of bugs successfully reproduced in FuzzBench (and not the number of bugs originally reported by OSS-Fuzz).

We run 30 fuzzing campaigns or trials (from here on, *runs*) for each $(fuzzer, benchmark)$ pair. We use the default seeds included by the fuzz target integrators in OSS-Fuzz (which is sometimes none), and set each campaign to 23 hours. This length was chosen to lower the cost incurred in running the experiment on the Google Cloud Platform. We deemed this an acceptable compromise since results in the literature often suggest using 20-24 hours as a stoppingc riterion [15].

A. Research questions

In this section we describe the motivations for our research questions and the methodology adopted to answer them.

1) RQ1 - Effectiveness: How do the chosen fuzzers compare against each other in finding bugs? Because the purpose of fuzzers is finding bugs in the SUTs, we first and foremost evaluate and compare fuzzers based on their bug finding ability.

We answer RQ1 by comparing fuzzers based on the following three metrics: number of *distinct crashes*[1]; number of distinct ground truth bugs found (i.e., *ground truth bug hits*); number of distinct crashes found that are not in the ground truth (referred to as *additional crashes*). From now on, whenever we use the term crashes without any adjective, we refer to crashes that have already undergone a de-duplication process run by FuzzBench.

We also compare fuzzers based on the total number of crashes they have found over all the trials, and on the number of crashes that only one fuzzer has been able to find across all the trials. The first metric offers insight into which fuzzer has the potential for finding a larger number of different bugs in a SUT, and to understand which fuzzer would be better suited to a fuzzing campaign composed of multiple short, potentially parallel, instances. The second metric offers interesting insights on which fuzzers are able to find bugs no other fuzzer can find, which is a characteristic to be factored in when designing a fuzzing campaign with multiple fuzzers, to avoid finding duplicated bugs. Additionally, we analyse how the fuzzer behaviour progresses over time during the fuzzing campaign.

In order to perform a robust analysis, we also devise four different approaches, based on statistical significance analysis and effect size test, to rank the fuzzers, as described in the following.

We define the ranking R adjusted based on the results of a given statistical significance test for a given fuzzer f and benchmark in Equation 1:

$$R(f) = \frac{1}{|EF_f|} \times \sum_{f' \in EF_f} NR(f') \qquad (1)$$

[1]For sake of space we report on the number of distinct crashes found only when they cannot be derived from the results of the other two metrics, as the number of distinct crashes can in most cases be computed as the sum of the number of the ground truth bug hits and the number of additional crashes.

where EF_f is the set of all fuzzers (f included) equivalent to f; and NR is the function that computes the ranking of a fuzzer for the desired metric. In summary, the ranking procedure assigns to a given fuzzer f the average ranking of all fuzzers equivalent to f. For example, if a fuzzer is ranked first since it found the highest number of crashes, but it is not statistically significant better than a fuzzer ranked second in terms of number of crashes, its adjusted ranking will be 1.5. We adopted this procedure to take into account not only of the difference between two fuzzers based on a given metric, but also the statistical significance of such a difference, thus providing a more robust ranking system.

To *rank the fuzzers per benchmark*, we use the Kruskal-Wallis statistical test [16] on the results obtained from the experiments pertaining to a given benchmark (considering each trial separately). If this test provides us with a significant result for a given benchmark, it signifies that at least two fuzzers behaved differently, and it will prompt us to run the Dunn's post-hoc test to determine which pair of fuzzers produced statistically significant different results.

To *rank the fuzzers across all benchmarks* we use the Friedman [11], [12] test, with a follow-up Nemenyi post-hoc test to determine which fuzzers are statistically equivalent and which ones are not. For the sake of space, more details on the process we use to run Kruskal-Wallis, Friedman and the post-hoc tests are provided in the online appendix [3].

We also rank the fuzzers based on the effect size analysisof the differences between the fuzzers' performance on each benchmark, by using the Vargha-Delaney $\hat{A}_{12}$ measure [33]. Based on this test, a statistical win is given to a fuzzer F_1 against a fuzzer F_2 when the value of the effect size measure $\hat{A}_{12}$ is strictly larger than 0.5; a loss is given when $\hat{A}_{12}$ is less than 0.5, a draw when $\hat{A}_{12}$ is exactly 0.5 (and no win is given to either fuzzer in this case). Moreover, a win is considered negligible (N) if $\hat{A}_{12}$ is less than 0.647, small (S) if $\hat{A}_{12}$ is less than 0.83, medium (M) if $\hat{A}_{12}$ is less than 0.974, and large (L) otherwise [14].

We rank the fuzzers not only based on the number of wins, losses and ties, but also on their respective magnitude (i.e., N,S,M,L). Since any kind of weighting given to negligible, small, medium and large wins could be considered subjective, we consider three different scoring systems: (i) "Linear Score", which assigns the weights $(3, 2, 1, 0)$ to L, M, S and N wins, respectively; (ii) "Quadratic Score" which assigns the weights $(4, 2, 1, 0)$ to L, M, S and N wins, respectively;(iii) "Adjusted Ranking' which first ranks each fuzzer based on the number of large, medium, small and negligible wins for a given fuzzer's tuple (L, M, S, N), and then adjusts such ranking based on Equation 1 considering as statistically equivalent the tuples that are either equal or *non-dominated*.[2]

[2]A tuple is said to dominate another if all the values in the first tuple are greater or equal than each corresponding value in the second tuple, and if there is at least one corresponding value greater than the other tuple.

2) RQ2 - Efficiency: How do the chosen fuzzers compare against each other in terms of efficiency? An important consideration in evaluating a fuzzer is how quickly fuzzers find bugs. Bugs that are found sooner can be fixed earlier, and possibly before deployment. We compare fuzzers based on the median expected time to first bug, and the median expected time to last bug, that is how much time it takes for them to find the first bug and how long they can continue finding bugs until they are not able to find anymore before the timeout. While it would be interesting to analyse other efficiency measures such as executions per second, memory and disk usage, none of the fuzzers currently integrated with FuzzBench reports additional statistics through their API, nor can FuzzBench reliably record any of such info. Thereby we could not analyse any additional efficiency measures.

3) RQ3 - Bug Types: Do certain fuzzers reliably find some type of bugs more than others? This aspect is particularly relevant for practitioners who have an interest in finding specific types of bugs that, for example, might be present in large quantities in a known codebase, or simply are not well covered by the existing test suite. If such a requirement exists, knowing which fuzzer is best suited to find which bugs could prove very beneficial. To answer this question, we categorise the bugs in the ground truth and the ones newly discovered by the fuzzers based on the bug-type provided by OSS-Fuzz or ClusterFuzz. We then check if there is any relationship between a given fuzzer and type of bugs. The description of the bug categories considered in this work can be found in our online appendix [3].

4) RQ4 - Coverage Correlation: Does coverage correlate with the number of crashes? Our last question aims at further analysing the claims made in previous work [15] that the usage of the coverage achieved as a proxy for found bugs is not optimal. Providing researchers and practitioners with additional and more robust empirical evidence answering such a question can guide them towards a more rigorous analysis of fuzzers in the future. Therefore we check for correlation between the coverage data gathered by FuzzBench and the number of crashes found. We do so in three different scenarios: across the entirety of the results, without any grouping; grouped per benchmark, in order to test if and which benchmarks are more prone to triggering actual bugs when their coverage is higher; grouped per fuzzer, in order to check whether the code coverage measure is a good predictor for the bug revealing abilities of a specific fuzzer. Since we have no guarantee that our data is normally distributed, we use the Spearman test [30] as our correlation test. The Spearman's ρ coefficient goes from -1 to 1, where -1 indicates a perfect inverse correlation, 1 a perfect direct correlation, and 0 a lack of correlation [30].

B. Benchmarks

To select the software systems to be used as a benchmark to assess and compare fuzzers, we have carefully analysed several options, including the CGC [7] and LAVA [9] benchmark suites, which, however, consist of synthetic programs and/or synthetic bugs, and the Fuzzer Test Suite [22], whose number of known bugs per benchmark was too low for our needs. We decided to use the OSS-Fuzz bug tracker [8] as a source for software systems under test, as it provides real-world data, a good amount of ground truth, and its software systems can be conveniently integrated in FuzzBench [27].

From OSS-Fuzz, we aimed at selecting projects with a sufficient number of bugs that can be reliably reproduced among three sets: (i) projects having the largest number of bugs; (ii) projects having the largest number of bugs that are difficult to find; (iii) projects chosen uniformly at random. We refer the reader to our online appendix [3] for the description of the collection procedure we performed to gather the subjects and the detailed list of benchmarks used in our evaluation.

C. Fuzzers

We investigate and compare nine fuzzers that are widely used in industry and previous work [13], [15], [27], and that are integrated with FuzzBench, in order to allow for reproduction and replication of our work: `AFL` [35], `AFLFast` [5], `AFL++` [10], `AFLSmart` [29], `entropic` [6], `Fairfuzz` [17], `honggfuzz` [19], `libfuzzer` [24], `mOpt` [25]. These fuzzers, excluding `honggfuzz` (which is a project developed independently by Google), can be roughly separated into two families: the *AFL family*, formed by `AFL`, `AFLFast`, `AFLSmart`, `AFL++`, `mOpt`, and `fairfuzz`, and the *libfuzzer family*, composed by `libfuzzer` and `entropic`. The fuzzers in each of the two families are a modified version of a same original fuzzer (i.e., AFL and libfuzzer, respectively).

D. Hardware

FuzzBench uses a single core preemptible Google Compute Engine instance to run each trial for a $(fuzzer, benchmark)$ pair, and a 96 core Google Compute Engine instance to run the process in charge of collecting the data regarding crashes, coverage, etc.. The data analysis was performed on a consumer PC with an AMD 3900x processor and 64GB of RAM.

E. Threats to Validity

We recognise the following possible threats to the validity of our study.

The chosen benchmarks could not be representative of the whole space of possible SUTs for general usage fuzzers. We acknowledge that our benchmarks have been chosen from a selection of open source projects voluntarily submitted to the OSS-Fuzz platform and have also been selected based on criteria that could introduce a bias. we consider the former threat unlikely as the projects in OSS-Fuzz cover a wide range of dimensions, complexity and usages, and the open source nature of the projects is needed for us to run an effective campaign on the SUTs, so commercial, closed source projects could not be used in this work. To account for the latter threat, we also include randomly selected SUTs in our benchmark (more info in our online appendix [3]).

In terms of construct validity, the ClusterFuzz's de-duplication technique, which we used to link the crashes with our ground truth, might not be completely accurate, and as such recognise different bugs as being the same one (overeager de-duplication), or vice-versa split one bug into two separate ones (insufficient de-duplication). ClusterFuzz, in addition to what it refers to as "de-duplication", actually uses another technique called "Grouping" for de-duplication which we do not use in our experiment. This technique uses Levenshtein distance to de-duplicate crashes that have slight variations in their stacks [1]. Because of this difference, our de-duplication may be less effective than ClusterFuzz's.

Finally, the bugs in our ground truth might favour `libfuzzer`, `AFL`, and `honggfuzz` since they were originally found by those fuzzers in OSS-Fuzz. Given the results shown in this paper, this seems unlikely, as `AFL` and `libfuzzer` were not ranked amongst the best and only a few bugs found by `honggfuzz` are in the ground truth. In order to mitigate such bias, we also analyse the results using "additional crashes", i.e., crashes not originally found by such fuzzers in OSS-Fuzz and ClusterFuzz.

IV. Empirical Study Results

In this section we present the results of our empirical study and the answers to our research questions. We refer to benchmarks by their project name for better readability. The two *php* benchmarks are referred to as *php-parser* and *php-execute* based on their target.

A. RQ1 - Effectiveness

Tables I and II present the ranking of fuzzers based on their effectiveness in finding bugs, while Tables III and IV report on the total number of bugs found and total number of unique bugs found by each fuzzer per benchmark, respectively. Additional results in terms of average number of crashes found by fuzzer per benchmark, and the number of crashes over time can be found in our online appendix [3]. We use the acronym GT to indicate the results regarding the ground truth bugs metrics, AC to indicate those related to the additional crashes (not in **GT**), and DC to indicate the results regarding the distinct crashes found as a whole (**GT + AC**).

Table I shows the rank of each fuzzer per benchmark, adjusted to account for statistical significance as described in Section III and Equation 1. The lower the rank, the better the fuzzer is positioned according to a given metric (i.e., DC, GT or AC) and the results of the Kruskal-Wallis statistical test (note the ranks are calculated independently for each metric).

Table II shows four different ranking approaches (differing for the type of statistical analyses done), which are instead calculated across all benchmarks. The first ranking approach, dubbed *Friedman Ranking*, ranks fuzzers according to the median number of crashes they found, and then adjusts these ranks based on Equation 1 and the results of the Friedman test as described in Section III-A1 (which, in our case, resulted in a p-value < 0.001). The other three ranking approaches (dubbed as *Effect Size Rankings*) are all based on the results

TABLE I: RQ1 – Ranking based on fuzzers performance per benchmark, adjusted based on Equation 1 and Kruskal-Wallis test results. The lower the rank, the better. Highlighted in blue (GT) and green (AC) the best performing fuzzer(s) per benchmark.

Benchmark	**AFL**		**AFLFast**		**AFL++**		**AFLSmart**		**entropic**		**Fairfuzz**		**honggfuzz**		**libfuzzer**		**mOpt**	
	GT	**AC**	**GT**	**AC**	**GT**	**AC**	**GT**	**AC**	**GT**	**AC**	**GT**	**AC**	**GT**	**AC**	**GT**	**AC**	**GT**	**AC**
arrow	3.5	3	3.5	3	3	3	3.5	3	8.5	9	7	6.5	8.5	8	4.38	6.5	3	3
ffmpeg	6.5	5	6.5	4.5	2	1.5	6.5	5	2.5	8	7	6.5	1.5	1.5	6	8	6.5	5
libarchive	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
matio	4.5	4.75	4.5	4.75	2.5	1.5	4.5	4.75	8.5	7.67	4.5	7.5	4	1.5	8.5	7.5	4.5	5.5
njs	5	5.31	5	5	5	3.4	5	5	5	4.43	5	6.25	5	6.25	5	5.31	5	4.43
openh264	4.5	6.5	5	7	4	2.17	4.67	6.5	4	1.5	9	7.75	4	3.5	4	3	4.5	6.5
php-execute	4	6	4	6	4	1.5	4	6	8.5	6	4	6	4	1.5	8.5	6	4	6
php-parser	5	4.5	5.44	5	5	4.5	5	4.5	4.71	4	5	3.67	4.88	4.5	6	8.5	5	5.86
poppler	5	3.5	5	3.5	5	4	5	3.5	5	8	5	8	5	3.5	5	8	5	3
proj4	6.5	7	6.5	7	6.5	4	6.5	7	1.5	1.5	6.5	7	2	1.5	2.5	3	6.5	7
stb	4	4.58	3.5	4.58	3.5	3.5	4	3.5	8.5	5.17	5.33	9	3.5	3.25	8.5	7.5	3.5	3.5
wireshark	5	5.5	5	5.5	5	5.5	5	4.93	5	3.5	5	5.5	5	9	5	1.5	5	4.93

TABLE II: RQ1 – Rankings based on fuzzers performance across all benchmarks, evaluated with the Friedman test, and the Vargha-Delaney effect size. The lower the rank, the better. Highlighted in red (DC), blue (GT) and green (AC) the best fuzzer(s) per benchmark.

Benchmark	**Friedman Rankings**			**Effect Size Rankings**								
				Linear Score			**Quadratic Score**			**Adjusted Ranking**		
	DC	**GT**	**AC**	**DC**	**GT**	**AC**	**DC**	**GT**	**AC**	**DC**	**GT**	**AC**
honggfuzz	1.5	1.5	1.5	1	1	1	1	1	1	1.5	1	2
AFL++	1.5	1.5	1.5	2	2	2	2	2	2	1.5	2	1
AFL	5	5.5	5.2	7	7.5	8	7	7.5	8	8	8.5	7
AFLFast	5	5.5	5.2	5	7.5	5	5	7.5	5	7	8.5	6
entropic	5	5.5	5.2	3	3	3	3	3	3	4	6.5	5
mOpt	5	5.6	5.2	4	4	4	4	5	4	3	3	3.5
AFLSmart	5	5.6	5.2	6	6	6.5	6	6	6	5	5	3.5
libfuzzer	8.5	6	8	8	5	6.5	8	4	7	6	6.5	8
fairfuzz	8.5	7	8	9	9	9	9	9	9	9	4	9

of the Vargha-Delaney effect size test. The first two (namely, *Linear Score* and *Quadratic Score*) assign a certain amount of points for each large, medium or small statistical win, and then rank the fuzzers based on their tally. The last ranking approach (namely, Adjusted Ranking) ranks the fuzzers based on their number of large, then medium, then small wins, and averages the ranks for those fuzzers that do not completely dominate another one (more details in Section III-A1). The results in terms of number of wins for each fuzzer, both against all other fuzzers on each benchmark, and against each other fuzzer on all benchmarks, can be found in our appendix [3].

Based on the results shown in Table I, we observe that AFL++ and honggfuzz are ranked top for more benchmarks than the rest of the fuzzers. While, from the results shown in Table II, we observe that according to the *Friedman ranking* both honggfuzz and AFL++ rank first for each of the metrics (i.e., DC, GT, AC). Whereas, according to the *Effect Size Rankings*, honggfuzz ranks first in all but one case (Adjusted Rankings for AC), and AFL++ ranks always either second (Linear Scores, Quadratic Scores, and Adjusted GT Rankings) or first (Friedman Rankings and Adjusted DC and AC Rankings). We also observe, based on the *Friedman rankings*, that two runner-up groups emerge: the one composed by *AFL*-based fuzzers, and the one by libfuzzer-based fuzzers, except for swapping Fairfuzz and entropic. While, the groups become indistinguishable based on the effect size rankings. The detailed results of the Friedman and Vargha-Delaney tests can be found in our online appendix [3].

Table III shows the number of total crashes found over all runs by each fuzzer per benchmark. Both AFL++ and honggfuzz rank first based on the number of ground truth bugs found, followed by mOpt. While, when considering the number of additional crashes, honggfuzz is the best fuzzer, while AFL++ ranks second. In addition to being first, honggfuzz also distinguishes itself from the other fuzzers when accounting for the raw number of bugs found. The efficacy of honggfuzz becomes more evident when we consider the ability of fuzzers to find crashes that no other fuzzer has found during our experiment. As shown in Table IV, honggfuzz ranks first (8 wins), while AFL++, AFLSmart entropic and libfuzzer rank second (3 wins each).

In terms of average number of crashes found across all three metrics and the corresponding statistically adjusted ranks, the best fuzzer is still honggfuzz, edging a win over AFL++. Besides, when we consider a broader picture including the results of the Vargha-Delaney rankings and the number of unique and total distinct crashes/bugs/additional crashes found (Table II), honggfuzz performs better than any other fuzzer in 54.17% of the effect size comparisons for distinct crashes, 39.58% for ground truth bugs, and 54.17% for additional crashes, getting ranked first 11 times and second only once.

B. RQ2 - Efficiency

Table V shows the rankings for all fuzzers across all benchmarks and metrics (i.e., DC, GT, and AC) based on the expected time to find their first and last crash.These ranks are not adjusted because despite the Friedman test gives a p-value

TABLE III: RQ1 – Number of bugs found by each fuzzer for a given benchmark. Each distinct bug is only counted once per (fuzzer, benchmark) pair. The first column shows the total number of bugs in the ground truth for each benchmark. Highlighted in green and blue the best fuzzer(s) per benchmark.

Benchmark	Bugs	AFL		AFLFast		AFL++		AFLSmart		entropic		Fairfuzz		honggfuzz		libfuzzer		mOpt	
		GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC
arrow	29	19	49	19	53	20	55	18	52	14	19	17	37	16	28	19	32	19	50
ffmpeg	35	4	16	5	24	17	30	4	22	9	13	3	15	25	48	5	7	5	20
libarchive	6	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0
matio	32	6	13	6	15	6	15	6	15	5	12	6	12	6	19	5	10	6	12
njs	6	1	4	1	4	1	5	1	4	1	5	0	4	1	3	1	3	1	4
openh264	4	3	3	3	2	3	6	3	4	3	9	3	0	3	5	3	6	3	3
php-execute	9	2	4	1	4	5	17	4	9	3	9	2	10	3	16	0	3	1	7
php-parser	6	3	8	2	7	3	9	2	7	4	10	3	9	2	10	1	2	1	9
poppler	13	3	48	4	47	4	47	4	46	4	39	1	28	1	53	1	15	6	52
proj4	27	0	0	0	0	6	10	0	0	16	21	0	0	22	62	14	11	0	0
stb	8	7	18	7	18	7	20	7	18	7	14	7	16	7	20	4	14	7	19
wireshark	8	1	7	0	5	0	7	1	8	0	8	0	8	0	0	1	7	1	7

TABLE IV: RQ1 – Number of bugs found by a given fuzzer that have not been found by any other fuzzer per benchmark. Highlighted in blue and green the best performing fuzzer(s) per benchmark.

Benchmark	AFL		AFLFast		AFL++		AFLSmart		entropic		Fairfuzz		honggfuzz		libfuzzer		mOpt	
	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC	GT	AC
arrow	0	1	0	2	1	2	0	2	0	0	0	0	0	0	1	0	0	0
ffmpeg	0	0	0	2	0	4	0	0	0	0	0	0	4	18	0	0	0	2
libarchive	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
matio	0	0	0	1	0	1	0	2	0	0	0	0	0	2	0	0	0	0
njs	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
openh264	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
php-execute	0	2	0	2	0	8	0	5	0	2	1	6	0	6	0	1	0	1
php-parser	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
poppler	0	3	0	1	0	2	0	1	0	3	0	3	0	9	0	2	1	2
proj4	0	0	0	0	0	1	0	0	1	0	0	0	7	40	0	0	0	0
stb	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
wireshark	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1	0	0

lower than 0.05, the Nemenyi tests only reveals a statistically significant difference in two cases (namely, the expected time to last distinct crash between `Fairfuzz` and `entropic`, and the expected time to find the last additional crash between `Fairfuzz` and `honggfuzz`). The raw data for the expected time to first/last crash can be found in our online appendix [3].

Since we analyse the expected time to find both first and last bug, it is interesting to observe that there are three main "behaviours" a fuzzer can assume, depending on its position in the rankings. Let us indicate with A a fuzzer's rank for the expected time to first crash, and with B the rank for the expected time to last crash, if $|B - A| \leq 2$ then we can say that the fuzzer performs in a manner inversely proportional to $avg(A, B)$ since the higher the values of A and B the shorter the time window in which the fuzzer has found its crashes. If $|B - A| > 2$ then the fuzzer either finds its first crash early and then finishes early too, or finds the first late and keeps finding crashes until the end; as such, we could say it behaves in a standard way, and its performance should be judged based on the user's goals.

It is clear that the best performing fuzzers for distinct crashes behave in the "standard" way, and as such should be considered situational. Conversely, for the other two metrics we have two clear winners and two runner-ups. Overall, `AFLFast` and `Fairfuzz` seem to be consistently highly efficient, while `honggfuzz`, `libfuzzer`, and `entropic` seem to consistently perform poorly. `mOpt`, while ranking first for the expected time to find first distinct crash (DC), and second for the additional crashes (AC), it does not show good results when considering its efficiency in finding ground truth bugs (GT).

C. RQ3 - Bug Types

We investigate whether some fuzzers exhibit different performance towards particular types of bugs by comparing the number of ground truth bugs and distinct crashes found by each fuzzer, categorised per bug type as reported by OSS-Fuzz and FuzzBench, respectively. For sake of space, the visual representation of these results can be found in our online appendix [3], while in the following we discuss the main trends observed.

The majority of all crashes found are *Integer-overflow*, *Undefined-shift*, *Divide-by-zero*, and *Heap-buffer-overflow* crashes, making up respectively 28.51%, 12.68%, 9.71%, and 8.58% of all the crashes. All fuzzers were able to find almost equally often *Integer overflows* bugs, except for `honggfuzz` and `AFL++`, which were notably able to find around 20 to 30 more crashes of this type (i.e., almost 35% more than the other fuzzers). Whereas `libfuzzer` found around half as many *Integer overflows* bugs than the average.

TABLE V: RQ2 – Ranking based on the average expected time to first crash and to last crash, across all benchmarks. The lower the better.

Benchmark	DC		GT		AC	
	First	Last	First	Last	First	Last
AFL	3	2	8	6	7	4
AFLFast	4	3	1	1	4	3
AFL++	7	6	2	2	5	8
AFLSmart	2	5	4	4	3	5
entropic	6	8	7	8	6	7
Fairfuzz	5	1	3	3	1	1
honggfuzz	8	9	6	7	8	9
libfuzzer	9	7	9	9	9	6
mOpt	1	4	5	5	2	2

Another interesting observation is the very large disparity in *Divide-by-zero* ground truth bugs found, ranging from the almost 84 bugs found by `honggfuzz`, to the single one found by `Fairfuzz`. All other fuzzers also find very few of this type of bugs, with the exception of `AFL++`, `libfuzzer`, and `entropic`, which manage to find between 30 and 40, but their results are still much lower than those achieved by `honggfuzz`.

We also noticed some minor differences. `Honggfuzz`, `AFLSmart`, `AFLFast`, and `mOpt` found a slightly higher number of *Abrt* bugs. Whereas, `Entropic`, `AFL`, `AFLFast`, `honggfuzz` and `libfuzzer` did not find any *Container-overflow* bugs (both in the ground truth and in terms of distinct crashes), while other fuzzers found only a single *Container-overflow* bug. However, it is possible that more bugs that could fall under this category were instead recorded as another similar type of bug.

Aside from the above differences, there are no relevant strong patterns in bug affinity. We also note that FuzzBench has a slightly different bug categorisation method from OSS-Fuzz, which we used for our ground truth. Thus, in some cases where fuzzers found zero ground truth bugs of a given type, the same fuzzers ended up finding additional crashes (not in the ground truth) of such a type. This is due to some of the bug types producing similar crashes, which makes it so that a same bug can be categorised in different ways, depending on the sanitiser, the fuzzer, or the environment in which it was found. Nevertheless, we mitigated this behaviour by considering the most prominent fault type when checking the bug types.

In summary, no particular relationship was found between fuzzers and bug types, other than *Integer-overflow* bugs for `honggfuzz` and `AFL++`, and *Divide-by-zero* for `honggfuzz`, `entropic`, `AFL++`, and `libfuzzer`.

D. RQ4 - Coverage Correlation

We discuss below the results of the correlation analysis carried out to answer RQ4. For the sake of space we report the raw correlation results in our online appendix [3].

The Spearman correlation test run on all data (i.e., without considering any grouping), reports a very weak correlation (Spearman's $\rho = 0.10$, p-value < 0.01), while the results for the data grouped by benchmark show different levels of correlation ($0.16 \leq \rho \leq 0.93$, p-value < 0.01), from weak to strong depending on the benchmark. When analysing the correlations by grouping the data by fuzzer, the results also tend to vary ($-0.14 \leq \rho \leq 0.31$, p-value < 0.01). We have also found two interesting cases: (`entropic` and `libfuzzer`) present a weak inverse correlation (entropic's $\rho = -0.07$, libfuzzer's $\rho = -0.14$) when considering results over all benchmarks, meaning that the more coverage, the fewer bugs found. A possible explanation for this is that the design of these fuzzers does not lend itself well to continuing fuzzing after a crash is found, as in the case of `libfuzzer` that was originally developed as an in-process fuzzer.

Overall, our results reveal that, while the overall coverage of all fuzzers on all benchmarks cannot be used as a strong predictor of bug discovery ability, for some benchmarks this is not the case. When we group the data by benchmark, many strong and significant correlations arise. Thus, not surprisingly, we can state that the correlation depends more on the SUT rather than on the fuzzer used.

E. Summary

The results of our empirical study in terms of effectiveness reveal that `honggfuzz` (especially) and `AFL++` are top ranked fuzzers, while the other AFL-based fuzzers lag slightly behind, and `libfuzzer` and `Fairfuzz` occupying the lowest ranks.

When considering the rankings based the Friedman statistical analysis, we are not able to find any statistically signifcant difference between the top fuzzers mentioned above. We did though find such a difference when considering the ranking based on Kruskal-Wallis test, and when analysing the effect sizes of the different results achieved by different fuzzers. This is partly in agreement with other experiments in the literature [15] that found no statistically relevant differences between `AFL`-based fuzzers when they are evaluated on a number of real-world found bugs. On the other end, our analysis shows that using effect size tests (which were not employed in any previous attempts at large scale fuzzer evaluations) together with using a larger number of samples can reveal some significant differences among fuzzers. In fact, we found that `honggfuzz` and `AFL++` are able to perform better than other fuzzers in our experiment, while `AFL++` shows similar results to the rest of the AFL-based fuzzers.

Overall our results provide researchers and practitioners with the following insights. If the most important metric is effectiveness in finding bugs, `honggfuzz` would be the best choice as it ranked first in almost all of our rankings. If for some reason `honggfuzz` is not a feasible choice, as an alternative we would recommend `AFL++`, or at least a member of the AFL family. If one is looking for a fuzzer with a focus on efficiency instead, either `Fairfuzz`, `AFLFast`, or `mOpt` are recommended. Finally, as a compromise between these two aspects (i.e., effectiveness and efficiency), we would recommend the use of `AFL++`.

V. Additional Observations

As can be seen in the plots available in our online appendix [3], the benchmarks we used have many bugs that take a long time to be found. In some cases these bugs took over 400 days to be found. Although this time can be due to several reasons, like an update to the fuzzing system that allowed the bug to be found after only a few hours while having been introduced many days prior, it is still a good proxy for the difficulty in finding the bug itself. Only five of our benchmarks were selected to target for a higher 75^{th} percentile of time to find bugs, but other benchmarks ended up being just as difficult as the ones we chose purposefully that way.

We can observe that most of the bugs found by the evaluated fuzzers are those with a time to find lower (and in most cases much lower) than 100 days, except for some outliers such as ffmpeg, proj4 and wireshark (see the second boxplot in our online appendix [3]). This suggests that future attempts at building an improved version of this benchmark suite should compute this metric beforehand and use it as a criterion to guide the choice of the benchmarks, in order to try avoid those with high time to find values (given our results, the cut-off point could be set at around 100 days, for ground truths sourced from OSS-Fuzz), or keep them to a minimum, as they provide limited information when used.

VI. Related work

A relatively small amount of work has been published on the topic of either fuzzer evaluation or large scale empirical fuzzer comparison, which has started gaining more and more attention since the work of Klees et al. in 2018 [15].

Klees et al. [15] analysed the evaluation methodology of 32 different fuzzer presented in published articles, and compared them with an ideal methodology that follows all known best practices and accounts for all possible parameters in a fuzzer evaluation. They found that no two analysed papers use the same methodology, and none of them fully follows even the subset of procedures encompassing at least the known best practices. They also suggested a list of basic criteria every fuzzer evaluation methodology should follow. Their paper has motivated our work, which follows their suggested guidelines wherever possible.

Li et al. [18] produced an (open-source) evaluation framework and runs an empirical comparison of eight fuzzers (which slightly overlaps with ours). Although the authors reached similar conclusions, their work tries to match crashes to CVEs as a ground truth.

The work by Paaßen et al. [28] evaluated mainly `AFL` derived fuzzers, with a strong focus on statistical analyses and robustness of the results. It evaluates fuzzers with different seeds and indeed presents, as claimed, the largest fuzzing experiment in terms of CPU-hours. As mentioned, though, it mostly focuses on the `AFL` family of fuzzers and uses as a benchmark suite a selection from the DARPA Cyber Grand Challenge (CGC) [7], a repository of synthetic (although real-world inspired) software with a single known bug per program. As such, although they do evaluate the fuzzers on 42 benchmarks, the number of real-world bugs actually involved in the experiments is only 42. In contrast, we evaluate fuzzers on a set of 183 real-world bugs.

Hazimeh et al. [13] evaluated seven fuzzers based on a novel methodology. The novelty consists mainly in the method they used to select the benchmark suite: A real-world program is selected and known and fixed bugs from previous versions are manually *forward-ported* in the most recent version of the executable. Extra code to allow for easy bug identification is also manually added. Although surely derived from real-world programs, we do not think the SUTs obtained this way can be considered proper “real-world” benchmarks, as the bugs end up being artificially reconstructed, with all the problems and nuances this process can introduce.

Tsuzuki et al. [32] performed a comparison of three fuzzers and reported their results in terms of path coverage. As it happens to most fuzzing comparison papers, this work does not use bugs as a source of information for quality and does not use multiple independent runs.

Our work differs from those discussed above mainly on the scale, rigour, and most importantly on the nature of the subject programs. We collected real-world subjects with known bugs found by ClusterFuzz within the OSS-Fuzz system. Moreover, we do not import bugs from other versions of the same software, but rather find the buggiest version through a rigorous procedure (as explained in the online appendix [3]). In this sense, we can state that the scenario in which the fuzzers are tested is as similar as possible to a real-world testing scenario. Furthermore, we performed multiple repeated runs of over 20 hours each to cater for the inherently stochastic nature of fuzzers, while also providing many metrics and analyses based on statistical tests.

VII. Conclusions and Future Work

We have carried out a rigorous and fair benchmarking of nine widely used and publicly available fuzzers based on real-world bugs. To this end we have implemented additional features in FuzzBench and manually curated a set of 183 real-world bugs from the OSS-Fuzz platform.

Our results show that `AFL`-based fuzzers and `honggfuzz` perform better in terms of effectiveness, while `libfuzzer`-based ones and `fairfuzz` perform better in terms of efficiency, with `honggfuzz` and `entropic` being the best fuzzers in terms of effectiveness and efficiency, respectively.

We found no relevant relationships between fuzzers and bug types, and only a weak correlation between region coverage metrics and the number of crashes/bugs found by a fuzzer. Whereas, we have found a strong correlation between coverage and number of crashes/found bugs in specific benchmarks, and as such we conclude that such correlation is SUT dependent.

Future work can extend our work with additional fuzzers and benchmarks based on the findings highlighted herein. To this end, we have made our code and data publicly available [2][3]. It would be also interesting to investigate why some fuzzers could find a different proportion of certain bug types, and if this behaviour is generalizable or due to chance.

Acknowledgement

Dario Asprone, Giovani Guizzo and Federica Sarro are supported by the ERC grant no. 741278 (EPIC).

References

[1] Abhishek Arya and Oliver Chang. Clusterfuzz: Fuzzing at google scale. *Blackhat Europe*, 2019.

[2] Dario Asprone. Fuzzbench - *Contributions*. https://github.com/google/fuzzbench/issues?page=2&q=author%3AVaush, October 2021.

[3] Dario Asprone, Jonathan Metzman, Abhishek Arya, Giovani Guizzo, and Federica Sarro. Additional files for "Comparing Fuzzers on a Level Playing Field with FuzzBench", 2022. `doi:10.5522/04/19249532`.

[4] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2021. `doi:10.1109/MS.2020.3016773`.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Procs. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, October 2016. `doi:10.1145/2976749.2978428`.

[6] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting fuzzer efficiency: an information theoretic perspective. In *Procs. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2020. `doi:10.1145/3368089.3409748`.

[7] DARPA. Darpa cyber grand challenge (cgc) binaries - *GitHub Page*. https://github.com/CyberGrandChallenge/, July 2020.

[8] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. March 2021. `arXiv:2103.11518`.

[9] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[11] Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937. `doi:10.1080/01621459.1937.10503522`.

[12] Milton Friedman. A correction. *Journal of the American Statistical Association*, 34(205):109–109, 1939. `arXiv:https://doi.org/10.1080/01621459.1939.10502372`, `doi:10.1080/01621459.1939.10502372`.

[13] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020.

[14] Melinda Hess and Jeffrey Kromrey. Robust confidence intervals for effect sizes: A comparative study of cohen's d and cliff's delta under non-normality and heterogeneous variances. *Paper Presented at the Annual Meeting of the American Educational Research Association*, 01 2004.

[15] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Procs. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[16] William H. Kruskal and W. Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952. URL: http://www.jstor.org/stable/2280779.

[17] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[18] 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. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security)*, pages 2777–2794.

[19] Google LLC. Honggfuzz - *Honggfuzz*. https://github.com/google/honggfuzz, December 2010.

[20] Google LLC. Clusterfuzz - *Clusterfuzz*. https://google.github.io/clusterfuzz/, July 2020.

[21] Google LLC. Fuzzbench - *Fuzzer Benchmarking As a Service*. https://google.github.io/fuzzbench/, July 2020.

[22] Google LLC. Google fuzzer test suite - *GitHub Page*. https://github.com/google/fuzzer-test-suite, July 2020.

[23] Google LLC. Monorail - *OSS-Fuzz issues*. https://bugs.chromium.org/p/oss-fuzz/issues/list, October 2021.

[24] LLVM. Llvm 10 documentation - *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html, July 2020.

[25] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[26] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[27] Jonathan Metzman, Lászlo Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Procs. of ESEC/FSE 2021*, page 1393–1403. ACM, 2021. `doi:10.1145/3468264.3473932`.

[28] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. My fuzzer beats them all! developing a framework for fair evaluation and comparison of fuzzers. In *Proc. of 26th European Symposium on Research in Computer Security*, volume 2021. Springer International Publishing, Darmstadt, oct 2021.

[29] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019. `doi:10.1109/TSE.2019.2941681`.

[30] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. URL: http://www.jstor.org/stable/1412159.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[32] Natsuki Tsuzuki, Norihiro Yoshida, Koji Toda, Kenji Fujiwara, Ryota Yamamoto, and Hiroaki Takada. A Quantitative Comparison of Coverage-Based Greybox Fuzzers. In *Procs. of the IEEE/ACM 1st International Conference on Automation of Software Test*, AST '20, page 89–92. Association for Computing Machinery, 2020. `doi:10.1145/3387903.3389304`.

[33] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[34] Michał Zalewski. Afl - *Tehcnical Details File*. https://lcamtuf.coredump.cx/afl/technical_details.txt, July 2020.

[35] Michał Zalewski. american fuzzy lop - *Home page*. https://lcamtuf.coredump.cx/afl/, July 2020.