On the Reliability of Coverage-Based Fuzzer Benchmarking

Marcel Böhme
MPI-SP, Germany
Monash University, Australia

László Szekeres
Google, USA

Jonathan Metzman
Google, USA

ABSTRACT

Given a program where none of our fuzzers finds any bugs, how do we know which fuzzer is better? In practice, we often look to code coverage as a proxy measure of fuzzer effectiveness and consider the fuzzer which achieves more coverage as the better one.

Indeed, evaluating 10 fuzzers for 23 hours on 24 programs, we find that a fuzzer that covers more code also finds more bugs. There is a *very strong correlation* between the coverage achieved and the number of bugs found by a fuzzer. Hence, it might seem reasonable to compare fuzzers in terms of coverage achieved, and from that derive empirical claims about a fuzzer's superiority at finding bugs.

Curiously enough, however, we find *no strong agreement* on which fuzzer is superior if we compared multiple fuzzers in terms of coverage achieved instead of the number of bugs found. The fuzzer best at achieving coverage, may not be best at finding bugs.

ACM Reference Format:
Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510230

1 INTRODUCTION

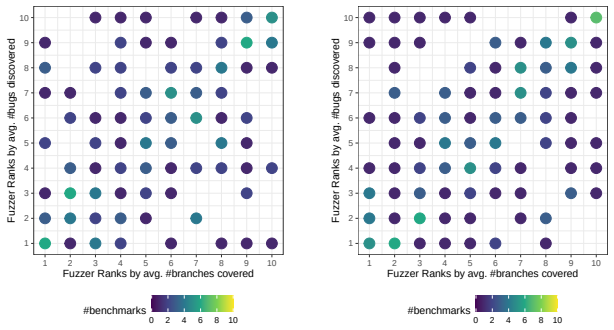
In the recent decade, fuzzing has found widespread interest. In industry, we have large continuous fuzzing platforms employing 100k+ machines for automatic bug finding [23, 24, 46]. In academia, in 2020 alone, almost 50 fuzzing papers were published in the top conferences for Security and Software Engineering [62].

Imagine, we have several fuzzers available to test our program. Hopefully, none of them finds any bugs. If indeed they don't, we might have *some* confidence in the correctness of the program. Then again, even a perfectly *non*-functional fuzzer would find no bugs in our program. So, how do we know which fuzzer has the highest "potential" of finding bugs? A widely used proxy measure of fuzzer effectiveness is the code coverage that is achieved. After all, *a fuzzer cannot find bugs in code that it does not cover.*

Indeed, in our experiments we identify a *very strong positive correlation* between the coverage achieved and the number of bugs found by a fuzzer. Correlation assesses the strength of the association between two random variables or measures. We conduct our empirical investigation on 10 fuzzers × 24 C programs × 20 fuzzing campaigns of 23 hours (≈ 13 CPU years). We use three measures of coverage and two measures of bug finding, and our results suggest: *As the fuzzer covers more code, it also discovers more bugs.*

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510230



(a) 1 hour fuzzing campaigns ($\rho = 0.38$). **(b)** 1 day fuzzing campaigns ($\rho = 0.49$).

Figure 1: Scatterplot of the ranks of 10 fuzzers applied to 24 programs for (a) 1 hour and (b) 23 hours, when ranking each fuzzer in terms of the avg. number of branches covered (x-axis) and in terms of the avg. number of bugs found (y-axis).

Hence, it might seem reasonable to conjecture that the fuzzer which is better in terms of code coverage is also better in terms of bug finding—but is this really true? In Figure 1, we show the ranking of these fuzzers across all programs in terms of the average coverage achieved and the average number of bugs found in each benchmark. The ranks are visibly different. To be sure, we also conducted a pair-wise comparison between any two fuzzers where the difference in coverage *and* the difference in bug finding are statistically significant. The results are similar.

We identify no strong agreement on the superiority or ranking of a fuzzer when compared in terms of mean coverage versus mean bug finding. Inter-rater agreement assesses the degree to which two raters, here both types of benchmarking, agree on the superiority or ranking of a fuzzer when evaluated on multiple programs. Indeed, two measures of the same construct are likely to exhibit a high degree of correlation but can at the same time disagree substantially [41, 55]. We evaluate the agreement on *fuzzer superiority* when comparing any two fuzzers where the differences in terms of coverage *and* bug finding are statistically significant. We evaluate the agreement on *fuzzer ranking* when comparing all the fuzzers.

Concretely, our results suggest a *moderate agreement*. For fuzzer pairs, where the differences in terms of coverage *and* bug finding is statistically significant, the results disagree for 10% to 15% of programs. Only when measuring the agreement between branch coverage and the number of bugs found and when we require the differences to be statistically significant at $p \leq 0.0001$ for coverage *and* bug finding, do we find a strong agreement. However, statistical significance at $p \leq 0.0001$ *only* in terms of coverage is not sufficient; we again find only weak agreement. The increase in agreement with statistical significance is *not* observed when we measure bug finding using the time-to-error. We also find that the variance of the agreement reduces as more programs are used, and that results of 1h campaigns do not strongly agree with results of 23h campaigns.

In summary, this paper makes the following *contributions*:

- ★ We introduce a novel methodology to evaluate proxy measures of fuzzer (or test suite) effectiveness. Specifically, we suggest evaluating agreement instead of correlation, and propose a bug-based evaluation without pre-determined ground-truth.
- ★ We provide the first evidence on the reliability of coverage-based benchmarking for the evaluation of fuzzer effectiveness. We confirm a very strong correlation and a moderate agreement.
- ★ We explore an interpretation of our results for *reaching* a fault versus *exposing* a bug (Section 6) and discuss our results in the larger context of fuzzer benchmarking, where we make concrete recommendations for future evaluations (Section 7).
- ★ We publish all data, the analysis, and the virtual experimental infrastructure. We provide precise instructions to reproduce and extend our experiments: https://doi.org/10.5281/zenodo.6045830

2 RELATED WORK

Code coverage has long been used as a proxy measure of the bug finding ability of a test suite. Fortunately, in practice the most common situation is that the test suite *detects no bugs*. Now, if all test cases pass, how do we assert whether the test suite is effective? Practitioners often rely on code coverage instead [33]. Underpinning coverage as a proxy measure is the insight that a test suite cannot find bugs in code that it does not cover. However, recent empirical studies on the correlation between code coverage and bug finding identify different degrees of correlation [9].

The *code coverage* of a test suite or fuzzer can be measured, e.g., as the number of program branches that are exercised by the test suite or fuzzer, respectively. The *bug finding ability* of a test suite (or fuzzer) can be measured, e.g., as the number of bugs found or the time it took to find the first bug. The *correlation* between two random variables measures the strength of their association and the direction of their relationship.

Using artificially injected bugs and developer-generated test suites, Inozemtseva and Holmes [32] find a *weak correlation* between coverage and test suite effectiveness when the size of the test suite is controlled for (and a moderate to strong correlation if test suite size is ignored). However, Chen et al. [9] raise concerns about the experimental methodology (i.e., the stratification of test set size) posing a significant threat to the validity of the results. Gopinath et al. [26] identified a *strong correlation* between code coverage and test suite effectiveness for developer-provided test suites and found the impact of test suite size neglible. For auto-generated test suites the correlation was *moderate to strong*; however, the majority of auto-generated test suites covered less than 20% of code while the coverage values for developer-generated test suites had a much wider spread, and they might have been written specifically for detecting these bugs. Gligoric et al. [22] find a *very strong correlation* between coverage and bug finding using different measures of correlation. *In contrast to this line of work*, we use real bugs instead of artificially injected bugs (i.e., mutants). Mutants may or may not be representative of real bugs [9, 35, 50]. Instead of developer-provided test suites, our study is concerned with "test suites" that were auto-generated by various fuzzers. In our study, test suite size is not a concern, either, as we explicitly control for the method by which the test suite (i.e., seed corpus) is generated.

Using real bugs and auto-generated test suites (generated by one fuzzer), Wei et al. [61] observe that the majority of bugs (>50%) are found in the last two thirds of the campaign when branch coverage increases only slightly from 90% to 94%. Along this qualitative reasoning, they conclude that "there is *weak correlation* between number of faults found and coverage". Kochhar et al. [37] find a *strong correlation* between coverage and bug finding for one program and a *moderate correlation* for another. However, Chen et al. [9] raise concerns about the correlation measure that was used and note that the association is likely stronger than indicated. More generally, Chen et al. expose several flaws in experimental methodologies of previous work and highlight common pitfalls in the statistical evaluation. Their own experiments indicate a *very strong correlation* between coverage and bug finding.

In our study, *we can confirm a very strong correlation*. However, *in contrast to all previous work*, we suggest the use of agreement instead of correlation for empirical investigations of test suite effectiveness. The *agreement* between two measures quantifies the degree to which both measures would agree on the relative performance of fuzzers. We define two types of agreement: agreement on *superiority*, which concerns two fuzzers; and agreement on *ranking*, which concerns more than two fuzzers. We say that two measures *agree on superiority* if both measures consider the same fuzzer better performing than the other, and the difference is *statistically significant*. We say that two measures *agree on ranking* if both measures order more than two fuzzers according to their average performance the same way, *not considering* statistical significance. Counterintuitively to the strong correlation result, we find that the agreement, both on superiority and ranking, is moderate.

Benchmarking bug finding tools is difficult. For static analysis tools, Dwyer, Person, and Elbaum [14] show that even small variations in the tool's configuration can give rise to a very large variation in the tool's bug finding effectiveness. For fuzzing, Gavrilov et al. [19] start from the observation that "bug-based metrics are impractical because (1) the definition of 'bug' is vague, and (2) mapping bug-revealing inputs to bugs requires extensive domain knowledge". In fact, we will elaborate on the challenges of bug-based evaluation in Section 7. Instead of counting the number of bugs, Gavrilov et al. [19] propose to measure the number of changes in program behavior over time that a fuzzer can detect.

To the best of our knowledge, our work is the first to evaluate whether coverage-based fuzzer benchmarking is reliable: Does the ranking of two or more fuzzers in terms of coverage agree with their ranking in terms of bug finding? The current guideline on sound fuzzer evaluation suggests that coverage-based benchmarking alone may be insufficient (referring to the contentious study [9] by Inozemtseva and Holmes [32] which suggests a weak correlation). Our study provides the first empirical evidence on the reliability of coverage-based fuzzer benchmarking.

3 EXPERIMENTAL SETUP

3.1 Research Questions

Our objective is to evaluate the degree to which a coverage-based and a bug-based benchmarking agree on fuzzer performance. We aim to answer the following research questions.

RQ.1 Correlation. How strong is the association between the coverage achieved by a fuzzer and its ability to find bugs?

RQ.2 Agreement. How strong is the agreement on the ranks or the superiority of the fuzzers in coverage-based versus a bug-based benchmarking?

RQ.3 Campaign Length. Does the agreement between coverage-based and bug-based benchmarking increase with the length of the fuzzing campaign? (Our default is 23 hours).

RQ.4 Campain Trials. Does agreement between coverage- and bug-based benchmarking increase with the number of campaigns per {fuzzer × program})? (Our default is 20 campaigns per combination).

RQ.5 Extrapolation Within one type of benchmarking, how strong is the agreement on the ranks or superiority of the fuzzers running 23 hour campaigns versus shorter campaigns?

RQ.6 Mitigation of Threats to Validity. (a) How strong is the agreement between two randomized rounds of coverage-based benchmarking? (b) How strong is the agreement between different measures of bug finding or between different measures of coverage? (c) How does agreement vary as the number of available programs increases?

3.2 Experimental Design

We evaluate these research questions using *a post hoc bug identification* instead of a pre-determined ground truth. While it requires substantially more effort, the post hoc identification allows us to avoid some of the pitfalls of ground-truth based benchmarking, as discussed in Section 7.[1] In our design, after conducting the fuzzing campaigns, we employ a process of automatic and manual deduplication to identify the unique bugs that each fuzzer discovered. Fuzzing campaigns may produce many bug reports, some of which actually pertain to the same unique bug. So, we sorted them out.

Our experiments generated 341,595 bug reports; too many for us to manually deduplicate. We used a variant of the Clusterfuzz deduplication approach to automatically group bug reports. After that, we manually deduplicated the 409 automatically deduplicated bugs to get 235 unique bugs. Two professional software engineers labeled the bugs to find duplicates. We note that our experimental design is indeed not very economically. Our fuzzers did not find a single bug in 30% (7/24) of the selected programs despite substantial fuzzing effort (Fig. 2). However, it allows us to mitigate a number of threats to validity of a ground-truth based evaluation (Sec. 7).

3.3 Fuzzers and Programs

Benchmark Details. The 24 *benchmark programs* we used are listed in Figure 2. Many of the programs are popular and well-maintained open-source software libraries that are widely used to support critical services in the internet. For instance, `libxml2` is a popular parser library for XML-documents, `php` is the interpreter for websites written in the PHP programming language, and `wireshark` is a popular network protocol analyzer. The set of benchmark programs ranges from parser libraries, protocol implementations, and implementations of compression algorithms all the way to OS service managers, interpreters, and platforms for in-memory analytics. Out of these 24 benchmark programs, there are seven (7) programs containing bugs that could not be found by any fuzzer, four (4) programs where bugs were very hard to find, and three (3) programs where no more more than two fuzzers could find bugs in at least one campaign.

[1] Examples are survivorship bias, observer-expectancy bias, and selection bias.

Name	Size	Harness Name	#Branches	#Known Bugs
libhevc	252.3k LoC	hevc_dec_fuzzer	54.7k	11
ndpi	58.0k LoC	fuzz_ndpi_reader	42.9k	15
libhtp	19.3k LoC	fuzz_htp	10.2k	1
aspell	28.1k LoC	aspell_fuzzer	28.4k	1
grok	26.6k LoC	grk_decompress_fuzzer	45.0k	4
matio	35.0k LoC	matio_fuzzer	46.7k	49
stb	70.2k LoC	stbi_read_fuzzer	6.7k	11
njs	89.0k LoC	njs_process_script_fuzzer	34.0k	10
zstd	93.2k LoC	stream_decompress	22.6k	1
openh264	140.1k LoC	decoder_fuzzer	39.3k	22
libgit2	224.9k LoC	objects_fuzzer	114.2k	3
poppler	225.6k LoC	pdf_fuzzer	184.7k	17
libxml2	505.1k LoC	xml_reader_file_fuzzer	101.8k	3
arrow	769.4k LoC	parquet-arrow-fuzz	473.9k	34
php	2.6M LoC	fuzz-execute	324.5k	9
php	2.6M LoC	fuzz-parser-2020-07-25	436.8k	8
wireshark	4.3M LoC	fuzzshark_ip	526.3k	10
proj4	168.2k LoC	standard_fuzzer	92.2k	36
tpm2	48.6k LoC	execute_command_fuzzer	20.8k	11
file	16.7k LoC	magic_fuzzer	8.6k	1
muparser	34.2k LoC	set_eval_fuzzer	7.6k	0
usrsctp	92.0k LoC	fuzzer_connect	53.6k	0
libarchive	166.5k LoC	libarchive_fuzzer	38.7k	10
systemd	588.4k LoC	fuzz-varlink	63.1k	1
Total	**13.2M LoC**		**2.7M**	**268**

Figure 2: Details about benchmark programs. In our data analysis, we excluded programs below the line because no more than two (of ten) fuzzers found a least one bug in at least one campaign.

AFL [63]	AFL++ [15]	AFLSmart [53]	AFLFast [4]
FairFuzz [38]	Eclipser [10]	MOPT [40]	Honggfuzz [58]
LibFuzzer [56]	Entropic [3]		

Figure 3: The fuzzers available in Fuzzbench, that we used in our experiments.

Benchmark selection. Our benchmark programs have been randomly selected from programs in OSS-Fuzz that have historically contained a relative high number of bugs. *OSS-Fuzz* [24] is a service that provides fuzzing for 500 open source projects. Integrations are usually performed by project maintainers and/or security researchers who write fuzz targets and compile seed corpora and dictionaries for the projects. This means that our benchmark programs have been prepared for fuzzing by the maintainers and not by us, which reduces experimenter bias. OSS-Fuzz automatically reports each bug it finds together with the first and last program version in which the bug exists. Most program versions do not contain any bugs, which is why a random selection of program versions from OSS-Fuzz would be prohibitively expensive. Most fuzzers would not find any bugs. Hence, we use the information from OSS-Fuzz to randomly chose our benchmark programs from program versions which are known to have an increased number of bugs. Similarly to OSS-Fuzz, all programs are instrumented with AddressSanitizer [57] as oracle to detect bugs. All fuzzing campaigns are started from an initial seed corpus provided from OSS-Fuzz.

Fuzzers. Figure 3 shows the list of fuzzers we used. We chose these fuzzers based on their importance and ease-of-use. Entropic, libFuzzer, Honggfuzz, AFL, and AFL++ are widely-used in industry while AFLSmart, AFLFast, FairFuzz, Eclipser, and MOpt-AFL are important academic works and extensions of AFL.

3.4 Variables and Measures

Our objective is to evaluate the degree to which a coverage- and a bug-based benchmarking agree on fuzzer performance. We have one main and two supplementary measures of coverage plus two main and one supplementary measure of bug finding.

Measures of Coverage. Our main measure of coverage is *branch coverage*, i.e., the number of branches in the program that the fuzzer has exercised until this point in the campaign. Branch coverage captures the control-flow in a program, subsumes statement coverage [22], is considered to be the most effective proxy measure of bug finding [20, 22], and is the conventional measure of coverage to evaluate coverage-guided greybox fuzzing [36]. Fuzzbench measures "region coverage" [12] in 15-minute intervals on a dedicated *measurer* instance[2] using `clang` compiler flags `-fprofile-instr-generate` and `-fcoverage-mapping` and the `llvm-cov` tool [12].

As supplementary measures of coverage, we also analyze the number of unique paths and the number of unique edges as measured by the AFL-fuzzer. The *number of unique paths* (#paths) continues to be a common performance measure for greybox fuzzers [17, 18, 68] despite its obvious flaws [36, 38]. The *number of unique edges* (#edges), reported as map size by AFL-based fuzzers, is often used as a proxy for branch coverage. AFL maintains a fixed-size hashmap containing an entry for every tuple of conditional jumps that are sequentially exercised in the program. For all measures of coverage, we directly evaluate coverage on the buggy program to avoid the clean program assumption [8].

Measures of Bug Finding. Our main measures of bug finding are *bug coverage*, i.e., the number of bugs that the fuzzer has found until a given point in the trial, and the *time-to-error*, i.e., the length of the fuzzing campaign when the first bug was found. In order to count the number of bugs (#bugs) at a particular point in time, we execute all bug-revealing inputs and remove all duplicates. Our method of deduplicating bugs is similar to ClusterFuzz's. For each crash reported in a trial, we take the crash type (e.g., "Heap-buffer-overflow") and the top three symbolized stack frames reported by AddressSanitizer or UndefinedBehaviorSanitizer. Crashes with the same type and stack frames are considered duplicates, and only one of them is counted. To further improve the quality of the deduplication, we manually removed the remaining duplicates. In order to measure the time-to-error (TTE), we report the length of the fuzzing campaign when the first crashing input was generated.

As supplementary measure of bug finding, we also count the *number of unique crashes* (#crashes), i.e., the number of "unique paths" that are exercised by crashing inputs. The number of unique crashes, similar to the number of unique paths, is a standard but contentious measure of bug finding. Crashes are flagged as such by standard code sanitizers, such as ASAN [57]. For both measures of bug finding, we directly evaluate bug finding on programs containing real bugs to mitigate threats to construct validity.

[2]Each fuzzing campaign runs on separate compute instance, called *runner*.

Spearman's ρ	Interpretation
0.00 - 0.09	Neglible correlation
0.10 - 0.39	Weak correlation
0.40 - 0.69	Moderate correlation
0.70 - 0.89	Strong correlation
0.90 - 1.00	Very strong correlation

(a) Taken from Schober et al. [55]

Cohen's κ	Interpretation
0.00 - 0.20	No agreement
0.21 - 0.39	Minimal agreement
0.40 - 0.59	Weak agreement
0.60 - 0.79	Moderate agreement
0.80 - 0.90	Strong agreement
0.91 - 1.00	Almost perfect agreement

(b) Taken from McHugh [42].

Figure 4: Interpretation of Spearman's ρ and Cohen's κ.

3.5 Statistical Analysis

In order to investigate the relationship between coverage-based and bug-finding based measures of fuzzer performance, we compute correlation and agreement.

Correlation [55] assesses the strength of the association and the direction of the relationship between two random variables. We assess the correlation between a measure of coverage and a measure of bug finding using *Spearman's rank correlation*. We use Spearman's instead of the more common Pearson's correlation as Pearson's assumes a linear relationship while our scatter plots in Figure 6 indicate an exponential one. Since both variables are *continuous*, represent *paired observations*, and their relationship is *monotonic*, the assumptions for Spearman's correlation are met. The interpretation of Spearman's ρ is shown in Figure 4.a.

Inter-rater Agreement [59] assesses the degree of agreement between two raters of the same phenomenon. In our case, we measure the agreement between a coverage-based measure of fuzzer performance and a bug-finding-based measure of fuzzer performance on the ranking or superiority of a fuzzer. Since coverage and bug finding measure the same construct, i.e., fuzzer performance, the assumption for assessing agreement is met. Schober et al. [55] note that "two variables can exhibit a high degree of correlation but can at the same time disagree substantially". Bland and Altman [41] suggest that any two measures of the same construct should necessarily be strongly correlated, but may not strongly agree.

Agreement on Rank. In order to benchmark multiple fuzzers simultaneously, it might seem reasonable to establish a *ranking*, where the best fuzzer according to some measure is ranked highest (cf. Fig. 1). A fuzzer's ranking for a program and time stamp is based on the corresponding average for that measure across all (twenty) trials. We measure the agreement on the coverage-based and bug-finding-based ranks of a fuzzer using *Spearman's correlation* [55].

Agreement on Superiority. Unlike a pair-wise comparison, a ranking does not consider the statistical significance of the difference between any two fuzzers. Hence, we also measure the agreement on the *superiority* of a fuzzer over another when superiority is established according to a measure of coverage versus a measure of bug finding. Using Cohen's kappa κ and disagreement proportion d, we measure agreement on superiority for pairs of fuzzers *only* where the difference in terms of *both* measures is statistically significant ($p \leq \{0.05, 0.001, 0.0001\}$) for at least 10% of the programs (>= 3). We believe there is insufficient evidence for fuzzer pairs where differences are statistically significant for the less than 10% of programs. Given a fuzzer pair, the coverage- and bug-based evaluation each "rates" which fuzzer is superior. We measure the agreement on these ratings across (at least three) benchmark programs. We also consider a third method using Spearman's ρ.

Disagreement proportion d is easy to interpret. Given a pair of fuzzers where the differences in terms of coverage *and* bug finding are statistically significant for at least 10% of the programs, the *disagreement proportion* d gives the proportion of programs where both fuzzers are considered superior according to coverage or bug finding, respectively: $d = (1 - p_o)$.

Cohen's kappa κ a standard, more robust measure of inter-rater agreement which also takes into account the possibility of the agreement occurring by chance [42, 59]. Given the same pair of fuzzers, *Cohen's kappa* κ is computed as the difference between the relative observed agreement on the superiority of a fuzzer p_o and the hypothetical probability of chance agreement p_e divided by the complement of the probability of chance agreement: $\kappa = (p_o - p_e)/(1 - p_e)$. Cohen's interpretation is shown in Figure 4.b.

Spearman's rho ρ allows us to use *all* data points using an ordinal rather than a binary variable for superiority: `1` for the superior fuzzer, `-1` for the inferior fuzzer, and `0` where the difference is not statistically significant according to the given p-value.

Statistical Significance. The evaluate the statistical significance of the difference between two fuzzers, we report Mann–Whitney U test—following the recommendations by Arcuri et al. [1] on the evaluation of randomized algorithms and Klees et al. [36] on the evaluation of fuzzers. *Mann-Whitney* U is a nonparametric test of the null hypothesis that, for randomly selected values X and Y from two populations, the probability of X being greater than Y is equal to the probability of Y being greater than X.

3.6 Experiment Infrastructure

We used the FuzzBench fuzzer evaluation platform [45] to conduct our experiments. The system consists of a dispatcher (the "brain" of an experiment) and workers. The *dispatcher* dispatches jobs (a) to build fuzzers and benchmarks, (b) to start separate *worker* machines, each of which runs one fuzzing campaign for one {fuzzer × program} combination, (c) to measure the results of the fuzzing campaigns and save results to a central SQL database, and (d) to generate reports based on the measurement results.

A *measurement* consists of measuring code coverage and crashes. Many crashing inputs may reveal essentially the same bug. For this reason, we employ a simple deduplication strategy to assign crashing inputs to bugs they reveal (cf. Section 3.4). For more details on FuzzBench, we refer the interested reader to the article by Metzman et al. [45] which introduces the FuzzBench infrastructure.

Each fuzzing campaign is run inside an Ubuntu 16.04 docker container on an `n1-standard-1` virtual machine instance running on Google Cloud. Each instance has 1 virtual CPU core, 3.75 GB of RAM, and 30 GB of disk space available to use. By default, we run 20 campaigns of 23 hours for each {fuzzer × program}-combination.

3.7 Reproducibility

The FuzzBench fuzzer evaluation platform was designed to facilitate open science, rigorous evaluation, and reproducibility. Figure 5 shows the identifiers of the FuzzBench experiments for this paper. We also link the exact commit hash (i.e., version) of FuzzBench which fixes the *exact* versions of all fuzzers, all benchmark programs, and the entire experimental platform that was used for our experiments. Each FuzzBench report (available at the link below) describes precisely how our empirical analysis can be reproduced.

Experiment Identifier	FuzzBench Commit	Description
`2021-02-17-bug-paper`	`38e344fe`	20 runs of 23 hours, all fuzzers, all subjects
`2021-08-19-crash-s`	`db192b60`	30 runs of 23 hours, all fuzzers, 11 subjects
`2021-08-19-crash-s2`	`db192b60`	30 runs of 23 hours, all fuzzers, 11 subjects

Figure 5: Reproducibility. Our experiments can be reproduced using the exact same settings and version of our experiment infrastructure.

Data Availability. The data used for our evaluation can be downloaded at from the corresponding FuzzBench reports at

- https://www.fuzzbench.com/reports/<experiment-id>

where the experiment identifier is given in Figure 5.

Data Analysis. We make our data analysis script available as Jupyter notebook together with all generated tables and images at:

- https://github.com/icse22data

Archival. For long-term archival, we also publish the data and analysis script at the Zenodo research artifact archival platform.

- https://doi.org/10.5281/zenodo.6045830

4 EXPERIMENTAL RESULTS

RQ1. Correlation

We investigate whether a fuzzer that covers more code is also better at bug finding. We ask whether the coverage that a fuzzer achieves is also a good predictor of the number of bugs found.

Methodology.[3] To assess the correlation between coverage and bug finding, we prepare a scatter plot of the mean branch coverage over the mean number of bugs found across all fuzzing campaigns for each program at any point in time (Fig. 6). For all three measures of coverage, we also compute Spearman's rank correlation between the mean coverage achieved and the mean number of bugs found during the average fuzzing campaign for each {program x fuzzer} combination at any point in time (Fig. 7).

Results. Figure 6 visually depicts the relationship between both coverage and bug finding. The scatter plots often show an almost straight line, suggesting a *very strong correlation*. In fact, the strength of the association between coverage and bug finding is confirmed in Figure 7. For all three measures, we see an average Spearman's rank correlation above 0.90, which we interpret as a *very strong correlation* (cf. Figure 4).

> There is a *strong correlation* between the coverage a fuzzer achieves and the number of bugs it finds in a program. As a fuzzer covers more code, it also finds more bugs.

The scatter plot in Figure 6 also provides hints as to the functional relationship. The linear increase with the log-scale y-axis seems to suggests an *exponential relationship*: A linear increase in branch coverage yields an exponential increase in the number of bugs found. While counterintuitive at first, it is not actually surprising if we consider that most of the code has already been covered even at the start of the campaign. Each fuzzing campaign starts with a seed corpus that already covers much of the program, and we measure

[3] The exact procedure can be found in our data analysis script which we have made publically available.

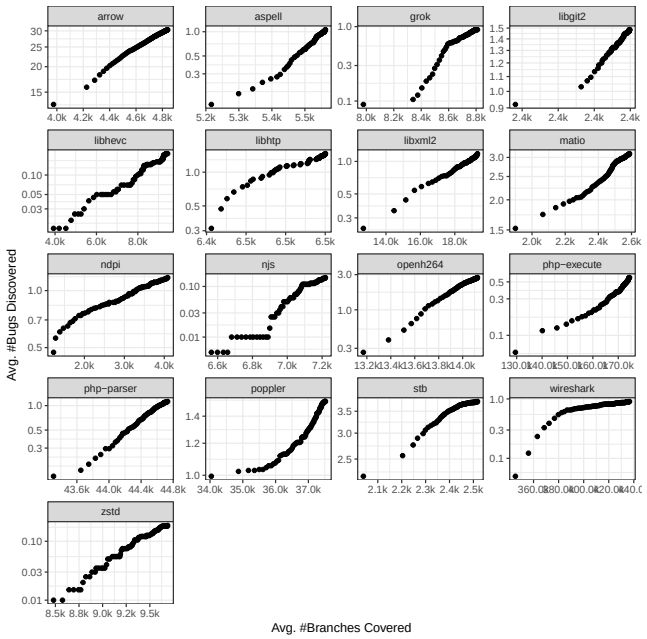


Figure 6: Scatter plot of the mean number of bugs found (on the log-scale) as the mean number of covered branches increases in the average fuzzing campaign for a benchmark.

the first data point after the first 15-minute interval, when most of the "shallow" branches have already been covered. This can be verified by looking at the start of the x-axis for each benchmark. We can see that the majority of branches which are covered in 23 hours have already been covered in the first 15 minutes. Covering a new branch gets harder over time. Even if coverage is fully saturated and not a single new branch can be covered, a fuzzer might still find new bugs. This interpretation agrees with the observation by Wei et al. [61] who found, for random testing, that the majority of bugs (>50%) were discovered in the last two thirds of the campaign, when branch coverage increased only slightly from 90% to 94%.

> In our study, there appears to be an exponential relationship between branch coverage and the number of bugs found.

RQ2. Agreement: Coverage versus Bug Finding

A strong correlation between two variables does not necessarily imply that they strongly agree [41, 55]. We investigate the degree to which the results of coverage-based benchmarking agree with the results of bug-based benchmarking.

Methodology (Ranking). For every {program × fuzzer × time stamp}-combination, we have twenty data points / trials. For every measure and for every {program × time stamp}-combination, we compute the fuzzer ranks by ordering all ten fuzzers according to the average measured value across all twenty trials. For every measure of coverage and every measure of bug finding, respectively, we compute the agreement between the coverage-based and bug-based ranking (in terms of Spearman's ρ).

	#Branches	#Paths	#Edges
arrow	0.999269	0.999276	0.999277
matio	0.990898	0.990896	0.990892
ndpi	0.888853	0.888625	0.888602
njs	0.918627	0.918636	0.918627
openh264	0.969526	0.969552	0.969522
poppler	0.949209	0.949217	0.949210
wireshark	0.888212	0.888212	0.888212
aspell	0.988724	0.988689	0.988703
grok	0.880887	0.880876	0.880710
libgit2	0.605309	0.600231	0.602031
libhevc	0.959148	0.959149	0.959147
libhtp	0.974873	0.965578	0.975135
libxml2	0.932176	0.932191	0.932172
php-execute	0.834285	0.834286	0.834285
php-parser	0.989402	0.989377	0.989400
stb	0.951317	0.951294	0.951250
zstd	0.830236	0.830244	0.830233
Average	*0.914762*	*0.913902*	*0.914553*

Figure 7: Average correlation (ρ) between coverage and #bugs found for all programs where at least one bug was found.

Methodology (Superiority). From ten fuzzers, we can construct 45 unique pairs of fuzzers. For each fuzzer pair, each program, and every measure, we determine effect size and statistical significance between both fuzzers in terms of mean and median of that measure across 20 trials of 23h. For each fuzzer pair, if the difference in terms of the coverage *and* in terms of bug finding is statistically significant at $p < \{0.05, 0.001, 0.0001\}$, for at least 10% of programs, we compute the agreement on superiority for this pair.

Results. Figure 8.a shows the agreement on ranking and superiority of a fuzzer in 23 hours campaigns. In terms of *ranking*, we observe a *moderate agreement* between coverage and bug finding.[4] In Figure 1 the moderate agreement is illustrated by the large spread. In terms of *superiority*, for Cohen's κ we observe a *weak to moderate agreement* for the average pair of fuzzers where the superiority along both measures is statistically significant. Across all measures, if we benchmark the average fuzzer pair using a coverage- versus bug-based approach, results disagree for 10% to 20% of programs. Figure 15 in the appendix shows a much lower agreement if we use the difference in *median* instead of the mean to establish superiority.

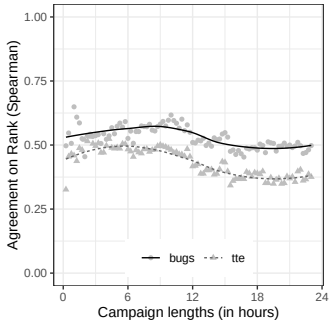
Only if the difference in terms of branch coverage *and* the difference in terms of the number of bugs found is statistically significant at $p \leq 0.0001$ (i.e., for 11 of 45 fuzzer pairs [24%]), we observe a *strong agreement* on the superiority of a fuzzer ($\kappa = 0.872$). In this case, a coverage-based and a bug-based evaluation of those eleven fuzzer pairs disagrees only for one benchmark (4.3%), on average. However, statistical significance at $p \leq 0.0001$ *only* of the difference in coverage is *insufficient*, we again only observe a weak agreement (see Figure 14 and Figure 9.d). The increase in agreement with statistical significance is *not* observed when we measure bug finding using the time-to-error (TTE).

> We observe a *moderate agreement* between a coverage-based and a bug-based benchmarking of fuzzer performance. For fuzzer pairs, where the differences in terms of coverage *and* bug finding is statistically significant, the results usually disagree for 10% to 20% of programs. Only for #branches versus #bugs, the agreement on superiority increases as the statistical significance for *both* differences increases.

[4] The interpretation of these values can be found in Figure 4.

	#Bugs			Time-to-Error		
	#Branches	#Edges	#Paths	**#Branches**	#Edges	#Paths
Ranking	$\rho = 0.498$	0.376	0.376	0.373	0.313	0.329
Superiority ($p \leq 0.05$)	$\kappa = 0.533$ (33%)	0.323 (27%)	0.535 (38%)	0.488 (31%)	0.322 (36%)	0.154 (33%)
Superiority ($p \leq 0.001$)	$\kappa = 0.327$ (24%)	0.450 (24%)	0.235 (24%)	0.612 (24%)	0.542 (22%)	0.485 (24%)
Superiority ($p \leq 0.0001$)	$\kappa = 0.872$ (24%)	0.428 (20%)	0.239 (24%)	0.553 (20%)	0.630 (22%)	0.496 (22%)
Superiority ($p \leq 0.05$)	$d = 0.113$ (33%)	0.213 (27%)	0.162 (38%)	0.211 (31%)	0.201 (36%)	0.360 (33%)
Superiority ($p \leq 0.001$)	$d = 0.141$ (24%)	0.144 (24%)	0.246 (24%)	0.177 (24%)	0.146 (22%)	0.166 (24%)
Superiority ($p \leq 0.0001$)	$d = 0.043$ (24%)	0.150 (20%)	0.262 (24%)	0.204 (20%)	0.082 (22%)	0.183 (22%)
Superiority ($p \leq 0.05$)	$\rho = 0.524$	0.374	0.424	0.367	0.421	0.274
Superiority ($p \leq 0.001$)	$\rho = 0.437$	0.397	0.319	0.375	0.376	0.379
Superiority ($p \leq 0.0001$)	$\rho = 0.448$	0.384	0.253	0.366	0.412	0.314

(a) Agreement on the rank (first row) and superiority of a fuzzer in 23hr campaigns in terms of Cohen's kappa (following three rows), disagreement proportion (middle three rows), and Spearman's correlation (last three rows). Each cell shows the measure of agreement and some cells, in parenthesis, the proportion of fuzzer pairs where the differences are statistically significant at the corresponding p-value ($p \leq \{0.05, 0.001, 0.0001\}$).



(b) Agreement on *ranks* between measures of bug finding and #branches after a campaign of x hours.

Figure 8: Agreement on ranking and superiority: Coverage versus Bug Finding

We also observe that the agreement on superiority is smallest for path coverage versus the number of bugs found, particularly for high significance thresholds. Path coverage has been a common performance measure for greybox fuzzers [17, 18, 68] despite its obvious flaws [36, 38]. The minimal agreement suggests abandoning path coverage as performance measure.

RQ3. Agreement Over Campaign Length

We investigate whether there is a suitable campaign length where a coverage-based and a bug-based evaluation maximally agree.

Methodology. To compute the agreement on ranking and superiority of a fuzzer over time, we followed the same methodology specified in the discussion for RQ2 for every of the 92 time stamps.

Results. Figures 8.b and 9 show the agreement on ranks and superiority over time, respectively. In terms of *ranking*, the agreement remains moderate over the entire duration. In the first nine hours, we observe an increase in agreement between an evaluation based on branch coverage versus one based on the number of bugs. However, the agreement on ranks decreases again, remaining moderate overall. In terms of *superiority*, we do *not* observe an increase in agreement (or a decrease in disagreement) over time for all three levels of statistical significance. The agreement between coverage- and bug-based benchmarking appears to decrease slightly. The differences are statistically significant for 20-30% of fuzzer pairs.

> In our study, we do *not* observe an increase in agreement (nor a decrease in disagreement) over time.

RQ4. Agreement Over Campaign Trials

We investigate whether there is a suitable number of campaigns per {fuzzer × program}-combination where the a coverage-based and a bug-based evaluation maximally agree.

Methodology. All results reported above are derived from our *default setup* where we run 20 campaigns of twenty three hours for each {fuzzer × program}-combination. In order to investigate, the agreement as the number of trials increases, we run an additional 40 campaigns for a subset[5] of the benchmark programs for a total of 60 campaigns of twenty three hours for each {fuzzer × program}-combination. From this set of 60 trials, we randomly sample n trials without replacement for each combination, where $n \in (1, 59)$, and compute agreement for those trials using the methodology specified in RQ2. To account for the randomness in the sampling, we repeat this experiment 50 times.

Results. Figure 10 shows the agreement on fuzzer ranking as the number of trials increases. For the first 20 trials in Figure 10.a, we can clearly see an increasing trend. As the number of trials increases, the agreement increases as well. However, from Figure 10.b, it seems that there is not much benefit in running more than 20 trials as the agreement increases only ever so slightly.

> The agreement between coverage-based and bug-based benchmarking increases as the number of campaigns increases. However, there does not seem to be much benefit in running more than 20 campaigns per {fuzzer × program}-combination.

RQ5. Agreement with Shorter Trials

We investigate the degree to which the results of (coverage-based or bug-based) benchmarking using shorter campaigns (say 1 hour) agree with the results of benchmarking using 23 hour campaigns.

Methodology. We measure the agreement on the ranking of a fuzzer when ranked at the end of the campaign versus earlier in the campaign, following the methodology we specified for RQ3.

Results. As we can see in Figure 11, there is a substantial difference in ranking when we rank fuzzers in a 1 hour campaign versus a 23 hour campaign. In fact, there is only *moderate agreement* between the results of a bug-based benchmarking at 1 hour versus those of a bug-based benchmarking at 23 hours. However, as we expect, the agreement increases with campaign length. In the bottom left of Figure 11.b, we can see that 15 minutes before the end of the 23 hour campaign, the ranks very strongly agree.

> The benchmarking results for rather short fuzzing campaigns may not strongly agree with results of sufficiently long campaigns. However, in our study, the benchmarking results for 12 hour campaigns do already *very strongly agree* with benchmarking results of 23 hour campaigns.

[5] Benchmark programs with 60 trials for each of the 10 fuzzers: arrow, libarchive, matio, ndpi, njs, openh264, poppler, proj4, tpm2, and wireshark.

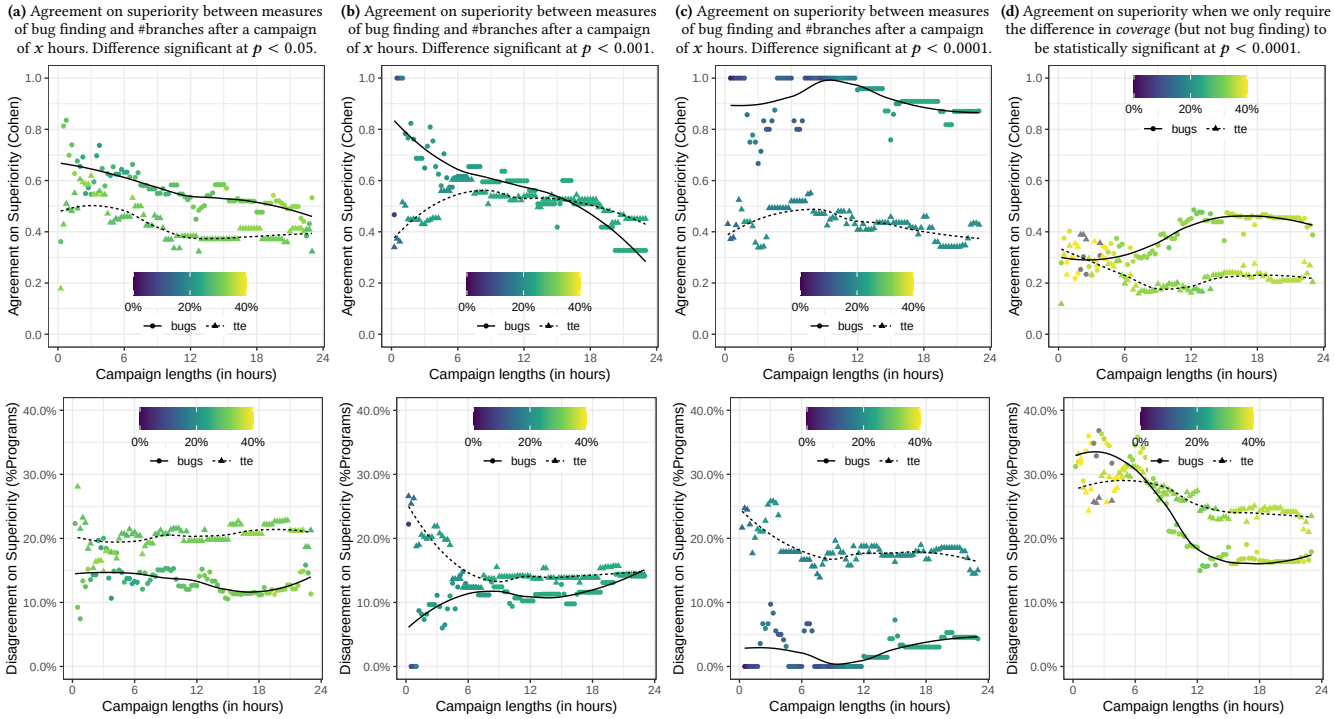
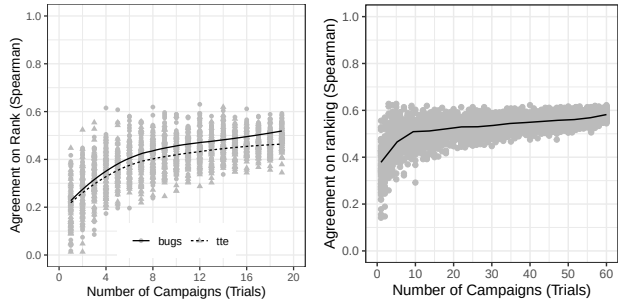


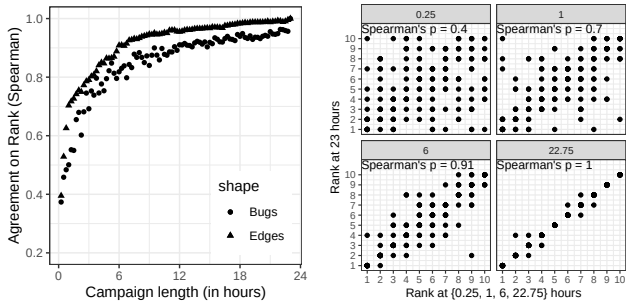
Figure 9: Agreement on superiority over campaign length. We show agreement when evaluating fuzzer performance based on branch coverage versus the number of bugs (solid line) and branch coveraged versus the time-to-error (dashed line). The color shows the percentage of fuzzer pairs for which the differences are statistically significant at the corresponding p-value ($p \leq \{0.05, 0.001, 0.0001\}$).



(a) Agreement over 20 trials for all programs.

(b) Agreement over 60 trials for a subset.

Figure 10: Agreement as the number of trials increases. The *solid line* shows the average agreement on the ranking of a fuzzer when ranked using branch coverage versus the number of bugs found. The *dashed line* shows the average agreement on the ranking of a fuzzer when ranked using branch coverage versus the time it takes to find the first bug (TTE).



(a) Agreement on ranks between a campaign of length x hours and one of length 23 hours. For instance, in terms of the number of bugs found, the rank of a fuzzer after 1 hour moderately agrees with the rank of that fuzzer after 23 hours.

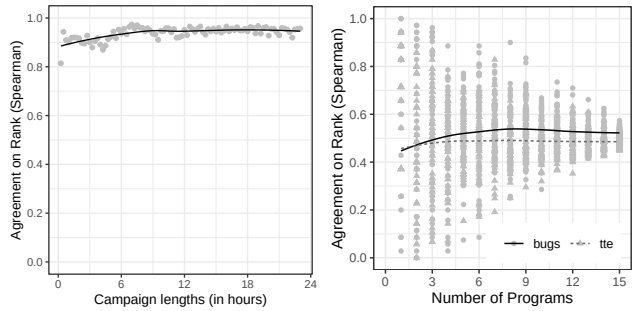
(b) Scatter plot of fuzzer ranks by #branches between a campaign of length $\{0.25, 1, 6, 22.75\}$ hours (facets) and a campaign of length 23 hours. Here, x are the ranks at the given campaign length and y are the ranks at 23 hours.

Figure 11: Agreement *within* coverage- or bug-based benchmarking as campaign length increases.

RQ6. Mitigations of Threats to Validity

We investigate several possible concerns and threats to validity.

(a) Baseline Agreement. A valid concern is that the results of coverage- and those of bug-based benchmarking may not agree simply because of some randomness in the measurement or broken measures of agreement. To investigate this concern, we check the baseline agreement between two random rounds of coverage-based benchmarking. From the 60 trials per {fuzzer × program}-combination generated for RQ4, we randomly sample 2×20 trials without replacement and compute agreement on ranks as specified in RQ2. To account for randomness, we repeat this experiment 50 times. To discharge the concern, we expect a high agreement. As we can see in Figure 12.a, we observe a *very strong agreement* on the rank of a fuzzer between two rounds of coverage-based benchmarking for every campaign length.



(a) Agreement between two randomized rounds of coverage-based benchmarking.

(b) Agreement as the number of benchmark programs increases.

Figure 12: Investigating threats to validity.

	#Bugs		#Branches	
	Time-to-Error	#Crashes	#Edges	#Paths
Ranking	$\rho = 0.671$	0.645	$\rho = 0.735$	0.647
Superiority ($p \leq 0.05$)	$\kappa = 0.868$ (31%)	0.893 (38%)	$\kappa = 0.785$ (44%)	0.647 (42%)
Superiority ($p \leq 0.001$)	$\kappa = 0.875$ (18%)	1.000 (20%)	$\kappa = 0.746$ (40%)	0.786 (38%)
Superiority ($p \leq 0.0001$)	$\kappa = 0.833$ (13%)	1.000 (20%)	$\kappa = 0.666$ (33%)	0.721 (33%)
Superiority ($p \leq 0.05$)	$d = 0.040$ (31%)	0.030 (38%)	$d = 0.094$ (44%)	0.083 (42%)
Superiority ($p \leq 0.001$)	$d = 0.042$ (18%)	0.000 (20%)	$d = 0.096$ (40%)	0.065 (38%)
Superiority ($p \leq 0.0001$)	$d = 0.056$ (13%)	0.000 (20%)	$d = 0.114$ (33%)	0.067 (33%)
Superiority ($p \leq 0.05$)	$\rho = 0.663$	0.757	$\rho = 0.666$	0.670
Superiority ($p \leq 0.001$)	$\rho = 0.634$	0.658	$\rho = 0.670$	0.703
Superiority ($p \leq 0.0001$)	$\rho = 0.556$	0.647	$\rho = 0.626$	0.669

Figure 13: Agreement among measures of bug finding (Column #Bugs) and measures of coverage (Column #Branches).

(b) Agreement between Measures. As discussed in Section 3.4, we have several measures of bug finding and several (supplementary) measures of code coverage. For a sound empirical analysis, we would expect that all measures of bug finding strongly agree and also that all measures of code coverage strongly agree along all our measures of agreement. As we can see in Figure 13, there is a *strong agreement* on superiority and ranking of a fuzzer when comparing fuzzers in terms of time-to-error versus counting the number of bugs found. Between measures of coverage, we identify a strong correlation in most cases, as well.

(c) Agreement Over Programs. Despite this being one of the largest empirical studies on the relationship between coverage and bug finding, a valid concern might be that the number of benchmark programs is relatively small. To investigate this concern, we randomly chose n programs without replacement out of the 17 programs where our fuzzers find bugs, and we compute agreement according to the methodology specified in RQ3, for $n \in (1, 17)$. To account for randomness, we repeat this experiment 50 times. Figure 12.b shows the scatter plot for the agreement on the randomly chosen programs as the number n of programs increases (grey dots and triangles), and the average agreement on fuzzer rank (solid and dashed line). As expected the average agreement is approximately constant as the number of programs increases. However, the variance is substantial, ranging between neglible and very strong agreement when only $n = 5$ benchmarks are chosen. However, at $n = 16$ benchmarks, the agreement ranges only within the moderate agreement band. Repeating this experiment by chosing programs *with* replacement gives similar results.

(a) The effect of randomness on the ranking within coverage-based benchmarking is neglible. (b) Even though measures of coverage do not agree with measures of bug finding, the measures agree within coverage-based and within bug-based benchmarking, respectively. (c) The number of benchmark programs has a substantial impact on our result. However, the variance in agreement is reasonably small for our benchmark size to support our conclusion, we would suggest. We do *not* recommend using less than 10 benchmark programs for coverage-based fuzzer evaluation.

5 THREATS TO VALIDITY

As for any empirical study, there are various threats to the validity of our results and conclusions.

One concern is *internal validity*, i.e., the degree to which our study minimizes systematic error. For our selection of fuzzers and benchmark programs there is a risk of experimenter bias, selection bias, survivorship bias, and confirmation bias. To minimize *experimenter* and *confirmation bias*, fuzzers and programs were prepared by independent developers. We picked programs randomly from the largest publicly available collection of fuzzer harnesses for 500 open source projects. Each harness was prepared by the corresponding maintainer. Each fuzzer was developed and added to FuzzBench either by the fuzzer developer or the FuzzBench team long before our study started. However, a possible cause of *survivorship and selection bias* is that – to keep experiment cost reasonable – the benchmark programs were selected from OSS-Fuzz such that a large number of bugs can be found. Many of those bugs were found by a subset of the evaluated fuzzers (e.g., AFL, AFL++, libFuzzer, Honggfuzz). However, our study is *not* concerned with establishing the state-of-the-art (finding which fuzzer is the best). Instead, we are investigating the reliability of coverage-based benchmarking, which mitigates most risk of selection and confirmaion bias.

Another concern is *external validity*, i.e., the degree to which our study can be generalized to and across other programs, fuzzers, bugs, and measures. To the best of our knowledge, ours is the largest study across all these dimensions. We chose a large variety of widely-used open-source C programs from different domains. Given the resuls in RQ6, we are confident that our results generalize to many more open-source C programs. We conduct our evaluation on a large number of actual bugs that these program contained organically some time in the past. We chose various, very successful greybox fuzzers which are used at Google [23, 24], Microsoft [46], other companies and many independent security researchers [15]. However, there is no guarantee that our results extend to (bugs in) programs written in other programming languages or fuzzers that are fundamentally different from greybox fuzzers. Even though our benchmark programs contain more known bugs than any other bug-based benchmark to-date, the number of bugs however is still low compared to e.g., the millions of branches in our benchmark programs (Figure 2). The sensitivity analysis in RQ6 on the impact of the number of programs chosen for the evaluation provides some confidence that our result extends to other, similar bugs. Therefore, it will be useful to replicate this study with other set of subject programs with real-bugs in them, preferably with an even larger and more diverse set of bugs.

A third concern is *construct validity*, i.e., the degree to which our study measures what it purports to be measuring. In this paper, we are interested in "fuzzer effectiveness", and one of the main questions we would like to answer is whether code coverage is a good metric for assessing it. We do this by comparing coverage metrics to bug-finding metrics, i.e., two of them: "number of bugs found" and "time to first bug found". Our assumption is that these bug based metrics are the ones that really capture fuzzer effectiveness. Among these two we believe that number of bugs is the more robust metric, as it is a more granular, give that it considers multiple bug data points, not just a single one. It is still possible, however, that due to our limited benchmark program set, which contains a limited set of bugs, the number of bugs that a fuzzer finds in *this* set is an imperfect metric (as discussed for the threat of the number of bugs on external validity). More specifically we measure "number of unique bugs found", where "unique" does not have an operational or universal definition. We rely on the OSS-Fuzz crash deduplication algorithm for this, which has been successfully field tested over many years. Our results for RQ6, where we assess baseline agreement and the agreement between measures provide further confidence in construct validity. We do not make the Clean Program Assumption [8] since coverage-based and bug-based benchmarking are conducted on the same program version.

Finally, *conclusion validity* relates to the reliability of our measurements and the validity of our statistical tests. We have addressed these issues by using well established standard methods to compute correlation, agreement and statistical significance. To triangulate, we use multiple measures (Section 3.4). We also carried out various sanity checks regarding agreement in Section 4 under RQ6.

6 DISCUSSION: REACHING A LOCATION VERSUS EXPOSING A BUG

The underpinning assumption of coverage-based benchmarking is that bugs that live in code that is not covered can also not be exposed. However, we find that the results of coverage-based benchmarking may not reliably indicate the results of bug-based benchmarking. So, how is reaching a certain location related to exposing a bug?

In our experiments, we use code sanitizers [11, 57] to detect bugs. During compilation, a *code sanitizer* injects assertions into the program binary that fail when, e.g., a memory safety issue occurs. So, covering those locations should be enough, right? Indeed, as Zhang and Mesbah [67] find that assertion coverage is strongly correlated with test suite effectiveness. Österlund et al. [49] demonstrate that a fuzzer that focusses on the coverage of sanitizer instrumentation outperforms existing fuzzers. Now, branch coverage *subsumes* "sanitizer coverage". Then, why do we not see a strong agreement between results of coverage-based and bug-based benchmarking?

If fuzzers were guaranteed to detect the bug when they reached the corresponding code location, then evaluating fuzzers based on code coverage *would be equivalent* to evaluating them based on bugs found. However, simply reaching a given branch or statement is often insufficient to trigger a bug. The root cause of a bug may not be localized in a single statement, but a certain sequence of statements may need to be executed throughout the code before the bug is exposed [6]. On the other hand, triggering the bug may be as hard as covering that program branch which reports that the bug has been triggered. Like bugs that cannot be exposed upon covering a branch, the coverage of that branch itself may already require a certain program state.

One hypothesis [64] is that faults could be empirically distributed in a non-uniform manner across the code base [47]. As future work, it will be interesting to investigate this and other hypotheses. Maybe we can find specific properties or differences between the typical program location (or branch) and fault locations or error conditions more generally. It would be interesting whether achieving these error conditions (versus achieving code coverage) require different capabilities from a fuzzer.

Yet, we still believe that code coverage is an excellent measurable objective function for a fuzzer. Coverage guidance has been the key to the recent success of greybox fuzzers [2]. Maximizing coverage is the key measurable objective in search-based software testing [43, 44]. Bugs are simply too rare to become an explicit objective or to provide a reasonable signal during fuzzing.

In our results, we see that the fuzzer that is better in achieving coverage may still be worse in finding bugs. The goal of this paper is to investigate how often we can observe this "asymmetry". If this happens rarely, that means that fuzzers can be soundly evaluated solely based on code coverage. If this happens often on the other hand, then it is recommended to use both code coverage and bugs to evaluate fuzzers.

7 FUZZER BENCHMARKING: CHALLENGES AND RECOMMENDATIONS

In 2020 alone, almost 50 fuzzing papers were published in the top conferences for Security and Software Engineering [62]. To ensure a realistic assessment of progress in the field, we need *sound measures of fuzzer effectiveness*. Only if our measures reflect a fuzzer's true bug finding ability, can we properly evaluate new tools against the state-of-the-art. Indeed, while improvements might seem reasonable, only a rigorous evaluation will tell for sure. For instance, ForAllSecure, the winning team at the DARPA Cyber Grand Challenge, burned one CPU-*year* every night to assess the previous day's improvements [48]. Nighswander adds that "many times 'obvious' changes made things worse and stupid things helped. Stats are vital". Towards this end, large benchmarking platforms have been built [28, 39, 45]; e.g., FuzzBench [45] has facilitated rapid and dramatic advances among the most successful fuzzers [31]. However, according to a recent survey of researchers and practitioners, sound fuzzer benchmarking remains a key open challenge [2].

In this paper, we provide the first empirical evidence that the results of a coverage-based evaluation are not strongly indicative of the fuzzers' relative bug finding ability. However, as we shall see next, a rigorous bug-based evaluation is not without perils, either.

7.1 Challenges of Bug-Based Benchmarking

Economic considerations. The most effective fuzzer finds the largest number of bugs. To evaluate the effectiveness of a fuzzer, in the perfect world, we would select a random, representative sample of programs (where we do not know whether any bugs can be found). However, we would quickly find that bugs are sparse in the typical program, and that the cost for experiments with a reasonable statistical power would be prohibitive.

Synthetic bugs. To make bug-based benchmarking more economical, researchers have proposed to articially inflate the number of bugs in these programs using *synthetic bugs* [7, 13, 51, 52, 54]. However, it is no final consensus on whether the synthetic bugs are realisitic [7, 21, 25]. In fact, as future work, we suggest to conduct a similar analysis of agreement, as proposed in this work, between benchmarking based on artificial bugs versus real bugs.

Ground truth. Alternatively, researchers have been *curating real bugs* that were historically found in programs [5, 6, 16, 27, 28, 34, 60]. While this approach is both economical and provides a more representative, objective ground truth, it is subject to several threats to validity that might not be obvious to the uninformed experimenter. **(a)** Evaluating fuzzers based on previously discovered bugs introduces a *survivorship bias*: Fuzzers that are better at finding previously *un*discovered bugs may appear worse than they are. On the other hand, fuzzers that contributed to the original discovery of some of the ground truth bugs may appear better than they are. **(b)** To increase the number of bugs in a program (and to reduce the benchmarking cost), curators may "front-port" several old bugs into one version. This introduces artificial *bug masking and interaction* effects, posing a threat to construct valdity. **(c)** To simplify bug counting and to provide the same bug oracle to all fuzzers, curators may manually translate each bug into a localized if-statement. This introduces an *observer-expectancy bias*. For instance, in this work, the relationship between coverage and bug finding is precisely the subject of our study (Section 6)?

Overfitting. Given a ground truth benchmark, researchers might be enticed to iteratively and unknowingly tune their fuzzer implementation to the bugs in the benchmark. Zeller et al. [65, 66] identify a particularly severe case of this confirmation bias which invalidates some empirical evidence in a well-cited paper. They recommend to augment bug-based evaluation with a coverage-based evaluation: "During testing, executing a location is a necessary condition for finding a bug in that very location. Since we are still far from reaching satisfying results in covering functionality, improvements in code coverage are important achievements regardless of bugs being found" [65].

7.2 Recommendations

For future evaluations of fuzzer performance, based on these results and our experience [45], we make the following recommendations. In the order of their appearance in the benchmarking process:

- **R1** If possible, select at least 10 respresentative programs. For each fuzzer-program combination, conduct at least 10 (better 20) campaigns of at least 12 (better 24) hours. Increasing these values improves generality and statistical power of the results.
- **R2** Select "real-world programs" that represent programs that are typically fuzzed in practice. Select "real-world bugs" that represent the set of bugs which are typically found in programs used in practice.[6] Improving the representativeness of the benchmark increases the external validity of the results. If experiment cost are a concern, authors can prioritize programs that (are likely to) contain a large number of bugs.
- **R3** Select as *baseline* the fuzzer that was extended to implement the technical contributions and make sure that the configurations (parameters, initial seeds, dictionaries, etc.) are equivalent. For instance, to demonstrate the advantages of structure-aware fuzzing [53], we would implement structure-aware fuzzing into a structure-*un*aware fuzzer and compare the extended against the baseline fuzzer. This improves construct validity and allows to attribute precisely the observed performance improvements to the proposed technical contributions. A comparison to other fuzzers may be conducted optionally if the authors wish to establish the new fuzzer as the new state-of-the-art. However, note that the observed improvements may be largely due to design and engineering differences (e.g., Honggfuzz versus AFL).
- **R4** Consider using a "training set" as benchmarks during the fuzzer development and a "validation set" possibly using an independent benchmarking platform for the actual empirical evaluation. This allows authors to reduce overfitting and confirmation bias.
- **R5** Measure and report both, coverage- and bug-based metrics to provide a holistic assessment of fuzzer performance. Use classical measures of coverage to facilitate (future) comparisons across various fuzzers. Do not use fuzzer-specific measures (such as AFL's number of paths). Use the same measurement tooling and procedure across all fuzzers and programs to increase internal validity. Consider using a post hoc bug identification (Section 3.2) rather than ground truth bugs to reduce threats to internal validity, such as survivorship bias.
- **R6** Assess and report various, non-parametric measures of effect size and statistical significance, such as Vargha-Delaney's $\hat{A}_{12}$ and Mann–Whitney U test, respectively [1]. This allows to quantify the magnitude of the differences and the degree to which the differences can be explained due to randomness.
- **R7** Discuss potential threats to validity and your strategies to mitigate the identified threats. For instance, discuss your strategies to mitigate selection, survivorship, observer-expectancy, and confirmation bias. If indicated, conduct an empirical evaluation of potential threats to validity.
- **R8** Report all specific parameters of the experimental setup (including how the programs, bugs, and initial seed corpus were chosen [30]), publish the tools (fuzzer and baseline) and the benchmark (programs and bugs) to faciliate the reproducibility of the results. Publish data, analysis, and figures to facilitate open access. Upload all artifacts to an open-access repository like Zenodo for long-term archival [29]. Reproducibility is the foundation of sound scientific progress.

[6] As future work, we suggest to evaluate the representativeness of synthetic bugs using a similar experimental setup as presented in Section 3.

ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of the Fuzzbench team. We also thank Stephan Lipp (TU Munich), Adrian Herrera (ANU), Mathias Payer (EPFL), and Rahul Gopinath (CISPA) for their feedback on earlier versions of this paper.

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing,*

	#Bugs			Time-to-Error		
	#Branches	#Edges	#Paths	**#Branches**	#Edges	#Paths
Superiority ($p \leq 0.05$)	$\kappa = 0.374$ (44%)	0.329 (44%)	0.418 (44%)	0.371 (44%)	0.217 (44%)	0.234 (44%)
Superiority ($p \leq 0.001$)	$\kappa = 0.360$ (40%)	0.240 (40%)	0.315 (42%)	0.305 (42%)	0.255 (38%)	0.177 (38%)
Superiority ($p \leq 0.0001$)	$\kappa = 0.388$ (33%)	0.204 (33%)	0.320 (42%)	0.336 (42%)	0.247 (38%)	0.101 (38%)
Superiority ($p \leq 0.05$)	$d = 0.227$ (44%)	0.241 (44%)	0.248 (44%)	0.251 (44%)	0.330 (44%)	0.313 (44%)
Superiority ($p \leq 0.001$)	$d = 0.208$ (40%)	0.247 (40%)	0.261 (42%)	0.241 (42%)	0.271 (38%)	0.304 (38%)
Superiority ($p \leq 0.0001$)	$d = 0.179$ (33%)	0.235 (33%)	0.273 (42%)	0.241 (42%)	0.265 (38%)	0.321 (38%)

Figure 14: Agreement on superiority when only requiring the difference in coverage to be statistically significant.

	#Bugs			Time-to-Error		
	#Branches	#Edges	#Paths	**#Branches**	#Edges	#Paths
Ranking	$\rho = 0.492$	0.378	0.361	0.376	0.315	0.324
Superiority ($p \leq 0.05$)	$\kappa = 0.312$ (33%)	0.038 (27%)	0.378 (38%)	0.233 (31%)	0.245 (36%)	0.151 (33%)
Superiority ($p \leq 0.001$)	$\kappa = 0.260$ (24%)	0.359 (24%)	0.102 (24%)	0.521 (24%)	0.331 (22%)	0.485 (24%)
Superiority ($p \leq 0.0001$)	$\kappa = 0.621$ (24%)	0.428 (20%)	0.025 (24%)	0.553 (20%)	0.476 (22%)	0.496 (22%)
Superiority ($p \leq 0.05$)	$d = 0.291$ (33%)	0.328 (27%)	0.293 (38%)	0.301 (31%)	0.309 (36%)	0.404 (33%)
Superiority ($p \leq 0.001$)	$d = 0.189$ (24%)	0.166 (24%)	0.309 (24%)	0.200 (24%)	0.203 (22%)	0.184 (24%)
Superiority ($p \leq 0.0001$)	$d = 0.097$ (24%)	0.150 (20%)	0.316 (24%)	0.204 (20%)	0.150 (22%)	0.183 (22%)

Figure 15: Agreement on ranking and superiority when considering the difference in terms of *Median* instead of the Mean.

Verification and Reliability 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

[2] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Opportunities. *IEEE Software* (2021), 1–9. https://doi.org/10.1109/MS.2020.3016773

[3] Marcel Böhme, Valentin Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 970–981. https://doi.org/10.1145/3368089.3409748

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

[5] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 105–115. https://doi.org/10.1145/2610384.2628058

[6] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? An Experiment with Practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/3106237.3106255

[7] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 716–730. https://doi.org/10.1145/3433210.3453096

[8] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 597–608. https://doi.org/10.1109/ICSE.2017.61

[9] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the Relationship between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 237–249. https://doi.org/10.1145/3324884.3416667

[10] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 736–747. https://doi.org/10.1109/ICSE.2019.00082

[11] LLVM Developers. 2021. Clang 13 documentation - UndefinedBehaviorSanitizer (UBSan). https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Online; accessed 16.Aug.21].

[12] LLVM Developers. 2022. Clang 15 documentation - Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#interpreting-reports. [Online; accessed 10.Feb.22].

[13] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the IEEE Security and Privacy (S&P'16)*. 1–10. https://doi.org/10.1109/SP.2016.15

[14] Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. 2006. Controlling Factors in Evaluating Path-Sensitive Error Detection Techniques. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. Association for Computing Machinery, New York, NY, USA, 92–104. https://doi.org/10.1145/1181775.1181787

[15] 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)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[16] Gordon Fraser and Andrea Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 178–188.

[17] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2577–2594.

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

[19] Miroslav Gavrilov, Kyle Dewey, Alex Groce, Davina Zamanzadeh, and Ben Hardekopf. 2020. A Practical, Principled Measure of Fuzzer Appeal: A Preliminary Study. In *20th IEEE International Conference on Software Quality, Reliability and Security, QRS 2020, Macau, China, December 11-14, 2020*. IEEE, 510–517. https://doi.org/10.1109/QRS51102.2020.00071

[20] Gregory Gay. 2017. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering*. Springer, 65–82.

[21] 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

[22] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Transaction on Software Engineering and Methodology* 24, 4, Article 22 (Sept. 2015), 33 pages. https://doi.org/10.1145/2660767

[23] Google. 2021. ClusterFuzz. https://google.github.io/clusterfuzz/. [Online; accessed 16.Aug.21].

[24] Google. 2021. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. [Online; accessed 16.Aug.21].

[25] Rahul Gopinath, Philipp Görz, and Alex Groce. 2022. Mutation Analysis: Answering the Fuzzing Challenge. arXiv:2201.11303 [cs.SE]

[26] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 72–82. https://doi.org/10.1145/2568225.2568278

[27] Péter Gyimesi, Béla Vancsics, Andrea Stocco 0001, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah 0001. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *Proceedings of the 12th International Conference on Software Testing, Verification and Validation*. IEEE, 90–101. https://doi.org/10.1109/ICST.

2019.00019

[28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. In *Proceedings of the joint international conference on Measurement and modeling of computer systems (SIGMETRICS'20)*. Association for Computing Machinery, New York, NY, USA, 29 pages. https://doi.org/10.1145/3428334

[29] Ben Hermann, Stefan Winter, and Janet Siegmund. 2020. Community Expectations for Research Artifacts and Evaluation Processes. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/3368089.3409767

[30] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795

[31] Marc Heuse. 2020. Twitter. https://twitter.com/hackerschoice/status/1302514351811842056. [Online; accessed 16.Aug.21].

[32] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[33] Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Association for Computing Machinery, New York, NY, USA, 955–963.

[34] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[35] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[36] 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 (CCS '18)*. Association for Computing Machinery, 2123–2138. https://doi.org/10.1145/3243734.3243804

[37] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 560–564. https://doi.org/10.1109/SANER.2015.7081877

[38] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[39] Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. 2022. FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer. In *Proceedings of the 44th International Conference on Software Engineering Companion (ICSE'22 Companion)*. 1–5. https://doi.org/10.1145/3510454.3516847

[40] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1949–1966.

[41] J. Martin Bland and DouglasG. Altman. 1986. Statistical methods for assessing agreement between two methods of clinical measurement. *The Lancet* 327, 8476 (1986), 307–310. https://doi.org/10.1016/S0140-6736(86)90837-8 Originally published as Volume 1, Issue 8476.

[42] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[43] P. McMinn. 2004. Search-based software test data generation: a survey: Research Articles. *Software Testing, Verification & Reliability* 14 (2004), 105–156.

[44] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 153–163. https://doi.org/10.1109/ICSTW.2011.100

[45] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[46] Microsoft. 2021. OneFuzz: A self-hosted Fuzzing-As-A-Service platform. https://github.com/microsoft/onefuzz. [Online; accessed 16.Aug.21].

[47] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/1315245.1315311

[48] Tyler Nighswander. 2020. Twitter. https://twitter.com/tylerni7/status/1374519171413766145. [Online; accessed 16.Aug.21].

[49] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2289–2306.

[50] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship between Mutants and Real Faults. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 537–548. https://doi.org/10.1145/3180155.3180183

[51] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 906–918. https://doi.org/10.1145/3468264.3468623

[52] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 214–225. https://doi.org/10.1145/2991079.2991103

[53] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019), 1–17.

[54] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/3236024.3236084

[55] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation coefficients: appropriate use and interpretation. *Anesthesia & Analgesia* 126, 5 (2018), 1763–1768.

[56] Kostya Serebryany. 2021. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. [Online; accessed 16.Aug.21].

[57] 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* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[58] Robert Swiecki. 2021. Honggfuzz. https://github.com/google/honggfuzz. [Online; accessed 16.Aug.21].

[59] Howard E.A. Tinsley and David J. Weiss. 2000. 4 - Interrater Reliability and Agreement. In *Handbook of Applied Multivariate Statistics and Mathematical Modeling*, Howard E.A. Tinsley and Steven D. Brown (Eds.). Academic Press, San Diego, 95–124. https://doi.org/10.1016/B978-012691360-6/50005-7

[60] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *ICSE*. IEEE / ACM, 339–349.

[61] Yi Wei, Bertrand Meyer, and Manuel Oriol. 2012. *Is Branch Coverage a Good Measure of Testing Effectiveness?* Springer Berlin Heidelberg, Berlin, Heidelberg, 194–212. https://doi.org/10.1007/978-3-642-25231-0_5

[62] Cheng Wen. 2021. Recent Papers Related To Fuzzing. https://github.com/wcventure/FuzzingPaper. [Online; accessed 16.Aug.21].

[63] MichałZalewski. 2021. american fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/. [Online; accessed 16.Aug.21].

[64] Andreas Zeller. 2022. A Tweet. https://twitter.com/AndreasZeller/status/1468142858553200644. [Online; accessed 10.Feb.22].

[65] Andreas Zeller, Sascha Just, and Kai Greshake. 2019. When Results Are All That Matters: Consequences. https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters.html. [Online; accessed 16.Aug.21].

[66] Andreas Zeller, Sascha Just, and Kai Greshake. 2019. 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. [Online; accessed 16.Aug.21].

[67] Yucheng Zhang and Ali Mesbah. 2015. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 214–224. https://doi.org/10.1145/2786805.2786858

[68] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Seyit Ahmet Çamtepe, and Yang Xiang. 2020. DeFuzz: Deep Learning Guided Directed Fuzzing. *CoRR* abs/2010.12149 (2020).