In Bugs We Trust? On Measuring the Randomness of a Fuzzer Benchmarking Outcome

ARDI MADADI[*], MPI-SP, Germany
SEONGMIN LEE[*], University of California at Los Angeles, USA
CORNELIUS ASCHERMANN, Ruhr-University Bochum, Germany
MARCEL BÖHME, MPI-SP, Germany

In Google's FuzzBench platform, we find that the outcome of coverage-based evaluation more strongly agrees with the outcome of a bug-based evaluation than an independent bug-based evaluation itself. Recently, Böhme et al. found that despite a very strong correlation between coverage achieved and bugs found, there is no strong agreement between the outcome of a coverage- and a bug-based evaluation: The fuzzer best at achieving coverage may be the worst at finding bugs. However, in trying to explain this moderate agreement, we wondered whether the outcome of bug-based benchmarking itself is perhaps much more "noisy" and turned to applied statistics to develop the tools necessary to investigate our hypothesis.

In this paper, we call this degree of "noisiness" of a benchmarking outcome the *concordance* of the benchmarking procedure and quantify it using a measure of statistical reliability widely used in psychology, called *mean split-half reliability*, i.e., the expected agreement on the benchmark outcome between two random halves of the benchmarking suite. In our experiments with FuzzBench and Magma, we find that the concordance of coverage-based benchmarking is consistently strong while that of bug-based benchmarking is weak on FuzzBench and moderate on Magma. In contrast to FuzzBench, for the Magma benchmark suite (which was designed for bug-based evaluation) a coverage-based evaluation does *not* predict the outcome of a bug-based evaluation better than an independent bug-based evaluation.

Moreover, to demonstrate the utility of concordance also for developers of benchmarking suites, we investigate concordance as a measure of benchmarking efficiency, as in green fuzzer benchmarking. We empirically confirm that the resources of a procedure with higher concordance can be reduced more substantially (in terms of campaign length or benchmark sampling size) while maintaining a similar benchmark outcome as a procedure with lower concordance. We report the corresponding savings in terms of carbon emissions.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Fuzzing, Mean Split-half reliability, Green Fuzzing, Internal Consistency, Fuzzer Rankings, Fuzzing Efficiency, Statistical Agreement, Statistical Correlation, FuzzBench, Magma

ACM Reference Format:
Ardi Madadi, Seongmin Lee, Cornelius Aschermann, and Marcel Böhme. 2026. In Bugs We Trust? On Measuring the Randomness of a Fuzzer Benchmarking Outcome. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE084 (July 2026), 22 pages. https://doi.org/10.1145/3797112

[*]These authors contributed equally to this work.

Authors' Contact Information: Ardi Madadi, MPI-SP, Bochum, Germany, ardi.madadi@mpi-sp.org; Seongmin Lee, University of California at Los Angeles, Los Angeles, USA, seongminlee@sigsoft.org; Cornelius Aschermann, Ruhr-University Bochum, Bochum, Germany, cornelius.aschermann@ruhr-uni-bochum.de; Marcel Böhme, MPI-SP, Bochum, Germany, marcel.boehme@acm.org.

ACM 2994-970X/2026/7-ARTFSE084
https://doi.org/10.1145/3797112

1 Introduction

Fuzzing continues to gain popularity in the industry due to its effectiveness and ease of adoption. As the field evolves, an increasing number of fuzzers continue to emerge–each claiming to best others in some performance metric. The most direct way to compare fuzzers is by measuring their effectiveness in finding bugs, i.e., *bug-based benchmarking*. However, a post-hoc analysis as in FuzzBench [33] requires substantial computational resources (after all, bugs are rare), while a ground-truth based analysis as in Magma [25] might feature various sources of bias [15]. Hence, a common approach remains *coverage-based benchmarking* where the coverage a fuzzer achieves serves as a proxy for the bugs it can find. After all, it can only find bugs in code that it executes.

Recently, with the release of new benchmarking suites like *Magma* [25] and the integration of bug-based benchmarks into *FuzzBench* [34], bug-based benchmarking has become a viable option. The latest version of Magma *forward-ports* 138 bugs across nine programs. Forward porting reintegrates and verifies the reproducibility of previously discovered bugs in stable–or even the latest–versions of these programs, ensuring they are compatible with various fuzzers. Meanwhile, FuzzBench counts bugs in a *post-hoc analysis* where the number of bugs found by a fuzzer is determined after the campaigns were run without any expectation of which bugs to find.

While both bug-based and coverage-based benchmarking evaluate fuzzer effectiveness, previous work [7] highlights an intriguing discrepancy between them: Although a very strong correlation exists between a fuzzer's ability to discover bugs and to achieve code coverage, the outcome of a coverage-based evaluation *does not strongly agree* with that of a bug-based evaluation. One possible explanation is that bugs are just much more sparsely distributed than coverage elements, which might lead to greater variability in the benchmarking outcome. This raises a key question: *"Does bug-based benchmarking exhibit a greater randomness than coverage-based benchmarking?"*

We refer to this randomness as the procedure's *concordance*. The concordance reflects how consistently a benchmarking procedure rates tool performance, or equivalently, the degree to which randomness influences the procedure's outcome. We quantify concordance using *split-half reliability* [11], which is commonly used in psychology to evaluate the statistical reliability of a test. The mean split-half reliability is the expected agreement on the procedure's benchmark outcome between two random halves of the benchmarking suite.[1] If the outcomes (dis)agree to a large degree, then the procedure is said to have a (low) high concordance.

Based on over 100,000 CPU hours ($\approx$ 11 CPU years) of fuzzing campaigns conducted using FuzzBench [34] and Magma [25], we analyze the concordance of bug- and coverage-based evaluation. For FuzzBench, we leverage the publicly available dataset created by Böhme et al. [7], which covers 20 fuzzing campaigns of 23 hours each for nine fuzzers applied to 24 C benchmarks, resulting in bug findings in 16 benchmarks. For Magma, we conduct our own experiment of a similar scale and closely aligned with the FuzzBench setup, comprising 20 fuzzing campaigns of 23 hours for each of eight fuzzers on 18 C benchmarks. We find a *strong concordance for coverage-based benchmarking*: The ranking of fuzzers in terms of branch coverage achieved is similar across independent benchmark subsets.

We only find a *weak to moderate concordance for bug-based benchmarking*. In FuzzBench, the split-half reliability of the ranking of fuzzers based on the number of bugs found in 23 hours is weak. Consider two disjoint, equi-sized benchmark subsets b_1 and b_2 randomly sampled from the FuzzBench benchmarking suite. The ranking of fuzzers in terms of bugs found on b_1 only weakly agrees with the ranking of fuzzers in terms of bugs found on b_2. Counterintuitively, the agreement is *stronger* when T are ranked on b_1 using coverage and T are ranked on b_2 using bug counts, compared to evaluating both subsets using bug counts alone. In other words, for FuzzBench, *a coverage-based*

[1] A *benchmark* is an element (e.g., a fuzz driver or a program) in a *benchmarking suite* (e.g., FuzzBench or Magma).

evaluation is more predictive of the outcome of a bug-based evaluation than an independent bug-based evaluation itself, due to that high degree of outcome randomness. This might explain the previous results by Böhme et al. [7] who observe that there is no strong agreement between the outcomes of coverage- and bug-based benchmarking in FuzzBench. For Magma, which is designed for bug-based evaluation, the concordance is *moderate*. Still, a coverage-based evaluation is equally predictive of the outcome of an independent bug-based evaluation. These findings establish coverage-based evaluation as a reliable benchmarking procedure.

Finally, to demonstrate the utility of concordance also for developers of benchmarking suites, we investigate *concordance as a measure of benchmarking efficiency*. As evident in our experiments, fuzzer benchmarking can have an extremely high carbon footprint. Green fuzzer benchmarking [35] is an effort to reduce the carbon footprint of fuzzer benchmarking substantially. We empirically confirm that the amount of resources necessary for a procedure with higher concordance can be reduced more substantially (in terms of campaign length or benchmarking suite size) while maintaining a similar outcome compared to a procedure with lower concordance. We provide the first model of the ecological footprint of fuzzer benchmarking and report the corresponding savings in terms of carbon emissions.

In summary, this paper makes the following contributions:

- We define the concept and measure of concordance for a benchmarking procedure and argue that a high-quality benchmarking procedure offers a high degree of concordance.
- We evaluate the concordance of bug-based and coverage-based benchmarking on FuzzBench and Magma, and find that the outcome of coverage-based benchmarking exhibits a substantially greater concordance. Notably, the outcome of a coverage-based evaluation is as effective as, or better than, a bug-based one in predicting the outcome of an independent bug-based evaluation.
- We establish concordance as a measure of benchmarking efficiency and show that the benchmark subset and campaign length for coverage-based can be reduced more substantially while maintaining a similar benchmark outcome as bug-based benchmarking.
- We make our code, data, and analysis publicly available. Full details for reproducing our experiments can be found at: https://github.com/ardier/in_bugs_we_trust/.

2 Concordance of a Benchmarking Procedure

Benchmarking procedure. The purpose of benchmarking is to compare two or more tools on a set of benchmarks in terms of their performance. A *tool* solves a particular problem where the *benchmark suite* is expected to represent instances of that problem. For instance, since fuzzers are designed to discover bugs in programs, one might define the benchmark suite as a collection of programs containing bugs. A *measure of performance* quantifies how well a tool solves a problem instance. For instance, one might measure the performance of a fuzzer in terms of the number of bugs it can find in a benchmark or, in their absence, the code coverage it can achieve. More formally, a *benchmarking procedure* $P = \langle B, m\rangle$ specifies the benchmark suite B and the measure of performance m.

Benchmarking outcome. Given a set of tools T, we define the *outcome* of a benchmarking procedure $P = \langle B, m\rangle$ as the ranking of T with respect to the performance measure m on the benchmark suite B, denoted by $outcome(P, T)$ or R:

$$outcome(P, T) = R = \{r^t\}_{t\in T}. \tag{1}$$

If there are only two tools $T = \{t_1, t_2\}$, we call this an analysis of superiority, and can also measure the effect size and statistical significance of the difference in performance of t_1 and t_2 to assess

Table 1. Interpretation of measures of agreement (and concordance), following Schober et al. [41].

Range	Interpretation	ID		Range	Interpretation	ID	
-0.00 to -0.09	Negligible disagreement	NNG	🟪	0.00 to 0.09	Negligible agreement	NG	⬜
-0.10 to -0.39	Weak disagreement	NWK	🟥	0.10 to 0.39	Weak agreement	WK	🟪
-0.40 to -0.69	Moderate disagreement	NMD	🟥	0.40 to 0.69	Moderate agreement	MD	🟦
-0.70 to -0.89	Strong disagreement	NST	🟥	0.70 to 0.89	Strong agreement	ST	🟦
-0.90 to -1.00	Very strong disagreement	NVS	🟥	0.90 to 1.00	Very strong agreement	VS	🟩

the impact of randomness across repetitions on the outcome. Throughout this paper, we report rankings over three or more tools, and suggest to break ties at random for an unbiased evaluation.

2.1 Agreement on Outcome

Given two benchmarking procedures P_1 and P_2, we can define their *agreement on the outcome* for a set of tools T as the degree of similarity between the corresponding rankings of T using P_1 and P_2, respectively. If we consider a benchmarking procedure as a rater of the tools' performance, then P_1's and P_2's agreement on outcome is also called *inter-rater agreement* [46].

Agreement vs correlation. Given two performance measures m_1 and m_2 and a benchmark set B, and a set of tools T, it is possible that the performance of $t \in T$ on B in terms of m_1 and m_2 is strongly correlated. Yet, procedures $P_1 = \langle B, m_1 \rangle$ and $P_2 = \langle B, m_2 \rangle$ may only moderately agree on the outcome for a set of tools T. As Schober et al. [41] highlight, "two variables can exhibit a high degree of correlation but at the same time disagree substantially." Similarly, Bland and Altman [4] note that two measures of the same construct may not always strongly agree, even if they correlate highly.

Example. Consider students' study time and exam scores. Although studying more generally leads to higher exam scores on average (high correlation), rankings based on study time and exam performance may not strongly agree. This could be especially the case when exam outcomes are influenced by stochastic factors, such as exam randomness or differences in study effectiveness (e.g., which topics the questions are drawn from), as well as students' conditions on the exam day.

Computing agreement. We compute the agreement between two rankings $R_1 = \{r_1^t\}_{t \in T}$ and $R_2 = \{r_2^t\}_{t \in T}$ of a set of tools T using Spearman's rank correlation or equivalently Pearson's correlation as the data are already ordinal [41]. The correlation coefficient ranges from -1 to 1. We interpret negative values as disagreement and positive values as agreement; the magnitude $|\cdot|$ indicates the strength of that (dis-)agreement. Table 1 summarizes this interpretation. Specifically, assuming a unique ranking for each tool in R_1 and R_2, we can compute the agreement $agreement(R_1, R_2)$ between two rankings R_1 and R_2 using Spearman's rank correlation coefficient, i.e.,

$$agreement(R_1, R_2) = 1 - \frac{6 \sum_{t \in T} \left(r_1^t - r_2^t\right)^2}{n_T(n_T^2 - 1)} \qquad \text{where } n_T = |T|. \tag{2}$$

Fuzzer benchmarking. In earlier work [7], we studied the agreement on outcome between coverage- and bug-based evaluation procedures for fuzzers T and benchmarks B in the FuzzBench benchmarking platform [33], e.g., $agreement(outcome(\langle B, \#edges \rangle, T), outcome(\langle B, \#bugs \rangle, T))$. Indeed, while the coverage achieved (#edges) and the number of bugs found (#bugs) are strongly correlated (i.e., a fuzzer that achieves more coverage also finds more bugs), the ranking of fuzzers in terms of coverage achieved and bugs found agrees only moderately (i.e., the fuzzer best at achieving coverage may be the worst at finding bugs). We used this result to propose reporting empirical results for both coverage- and bug-based benchmarking when evaluating fuzzer performance. However, what we seemed to miss is that *the outcome of a bug-based evaluation* itself *may be substantially noisy*: even two bug-based evaluations on disjoint, random, equally sized benchmark subsets may not strongly agree. This is precisely the subject of our study.

2.2 Concordance of a Procedure

We let the *concordance* of a benchmarking procedure $P = \langle B, m \rangle$ measure how "noisy" the outcome of P is when evaluating a set of tools using m over the set of all unique partitions of B into two disjoint, equi-sized subsets B_1 and B_2. Informally, concordance measures how reliably the outcome of P predicts the tools' performance on an unknown benchmark set of the same size. Treating a benchmarking procedure P as a rater of tool performance, we interpret the concordance of P as the *statistical reliability of* P, specifically its internal consistency.

Split-half reliability (γ). We measure the concordance $\gamma(P, T)$ of a benchmarking procedure P on tools T using the *mean split-half reliability* [13, 32, 48, 54]. Split-half reliability is widely used in psychology as a measure of the statistical reliability of a survey and indicates the extent to which survey questions on a scale consistently measure the same underlying concept.

The mean split-half reliability of the procedure $P = \langle B, m \rangle$ given tools T is the expected agreement in outcome between two procedures $P_1 = \langle B_1, m \rangle$ and $P_2 = \langle B_2, m \rangle$, where B_1 and B_2 range over all unique disjoint halves of the benchmark suite B, i.e.,

$$\gamma(P, T) = \frac{1}{|S_B|} \sum_{\langle B_1, B_2 \rangle \in S_B} agreement(outcome(\langle B_1, m \rangle, T), outcome(\langle B_2, m \rangle, T)) \tag{3}$$

where

$$S_B = \left\{ \langle B_1, B_2 \rangle \;\middle|\; (B_1, B_2 \subseteq B) \wedge (B_1 \cap B_2 = \emptyset) \wedge \left(|B_1| = |B_2| = \left\lfloor \frac{|B|}{2} \right\rfloor \right) \right\} \tag{4}$$

Technically, for a benchmark suite B of size $|B|$, $\gamma(P, T)$ only computes the expected reliability of subsets of size $\lfloor |B|/2 \rfloor$ rather than the entirety of the benchmark suite. While we do not use it in this paper (as it is a simple monotonic transformation and hampers an intuitive interpretation), the *Spearman–Brown formula* $2\gamma(P, T)/(1 + \gamma(P, T))$ can adjust the value to the full benchmark set of size $|B|$. The mean split-half reliability is also related to *Cronbach's alpha* [13], another widely used measure of reliability. Under the equal-variance assumption, alpha provides a conservative lower-bound estimate of the mean split-half reliability when scaled to the full benchmark suite [48].

As an aside, the split-half reliability defined over a particular split $\langle B_1, B_2 \rangle \in S_B$ (as originally conceived by Charles Spearman and William Brown [44]) is sensitive to how the benchmarks $b \in B$ are divided between B_1 and B_2. In our case, one half B_1 may contain benchmarks with a lot of bugs while the the other half B_2 may not contain any bugs at all. The *mean* split-half reliability reduces the influence of any particular split [13].

Proxy split-half reliability (ω). Intuitively, the mean split half reliability also measures how "predictive" the outcome on one half of the benchmark suite is of the outcome on the other half *using the same performance measure* m_{orig}. To evaluate how predictive the outcome using a proxy measure m_{proxy} is of the outcome using the original measure m_{orig}, in comparison, we define the proxy split-half reliability. We assume that m_{proxy} and m_{orig} both measure the same latent construct (i.e., fuzzer effectiveness) [16], which is also precisely what permits a comparison with split-half reliability [47]. Specifically, given two measures m_{proxy} and m_{orig}—if the split-half reliability for a m_{orig}-based evaluation is *lower* than the proxy split-half reliability of a m_{proxy} w.r.t. m_{orig}, then a m_{proxy}-based evaluation is *more predictive* of the outcome of a m_{orig}-based evaluation than an independent m_{orig}-based evaluation itself.

Example. Imagine the above situation arises when m_{proxy} is #edges (coverage achieved) and m_{orig} is #bugs (the number of bugs found). Then, it means that a coverage-based evaluation is more predictive of bug-based outcomes than an independent bug-based evaluation.

Given a set of tools T, a set of benchmarks B, and two measures m_{proxy} and m_{orig} of the tools' performance, we compute the *proxy split-half reliability* $\omega(m_{\text{proxy}}, m_{\text{orig}}, B, T)$ of a benchmarking procedure using m_{proxy} w.r.t. a procedure using m_{orig} as the expected agreement on outcome between two procedures $P_1 = \langle B_1, m_{\text{proxy}}\rangle$ and $P_2 = \langle B_2, m_{\text{orig}}\rangle$ using all unique, disjoint halves B_1 and B_2 of B, i.e.,

$$\omega(m_{\text{proxy}}, m_{\text{orig}}, B, T) = \frac{1}{|S_B|} \sum_{\langle B_1, B_2\rangle \in S_B} \mathit{agreement}(\mathit{outcome}(\langle B_1, m_{\text{proxy}}\rangle, T), \mathit{outcome}(\langle B_2, m_{\text{orig}}\rangle, T)). \tag{5}$$

where S_B is defined as in Equation (4). For efficiency, instead of evaluating over all possible splits in S_B, we approximate these reliabilities using maximum likelihood estimates on a random subset of S_B. Specifically, both $\gamma(P, T)$ and $\omega(m_{\text{proxy}}, m_{\text{orig}}, B, T)$ are estimated by $\hat{\gamma}$ and $\hat{\omega}$, which compute the average agreement over a random subset $S \subseteq S_B$ of a given size.

3 Experimental Setup

In this paper, we primarily investigate the concordance of bug-based and coverage-based benchmarking, and examine whether the outcome of a coverage-based evaluation can be a better predictor of the outcome of a bug-based evaluation than an independent bug-based evaluation itself.

Second, we explore the potential of our proposed concordance measure as a predictor of benchmarking efficiency. Specifically, we conjecture that, given sufficient effort, it is possible to substantially reduce the size of benchmark subsets for highly concordant procedures while incurring only a negligible impact on the benchmarking outcome.

The research questions and experimental setup for the first part are presented in this section, with corresponding results reported in Section 4. Under the same experimental setup, the second part is discussed in Section 5.

3.1 Research Questions

- **RQ.1** (*Concordance*). What is the concordance of a bug- versus a coverage-based fuzzer benchmarking procedure? In other words, how reliable is the outcome of a bug-based evaluation of fuzzer performance in comparison to that of a coverage-based evaluation?
- **RQ.2** (*Concordance as a Function of Benchmarking Suite Size*). How does the concordance of a benchmarking procedure behave as the size of the benchmarking suite increases?
- **RQ.3** (*Split-half Reliability versus Proxy Split-half Reliability*). How does the concordance of bug-based benchmarking compare to the agreement on outcome between coverage- and bug-based benchmarking? If proxy split-half reliability is higher, then the outcome of a coverage-based evaluation better predicts the outcome of a bug-based evaluation than an independent bug-based evaluation itself.

Open Science. We make both our datasets and analysis script available in section 9.

3.2 FuzzBench: Fuzzers and Benchmarks

FuzzBench [34] is a fuzzer benchmarking platform developed and computationally supported by Google to help the fuzzing community (incl. fuzzer developers, maintainers, and users) evaluate

Table 2. Programs used from FuzzBench and Magma datasets. #B is the number of benchmarks (fuzz drivers); Size is the number of lines of code (LoC); KB is the number of known bugs.

FuzzBench								**Magma**							
Name	**#B**	**Size**	**#KB**	**Name**	**#B**	**Size**	**#KB**	**Name**	**#B**	**Size**	**#KB**	**Name**	**#B**	**Size**	**#KB**
aspell	1	30.0k	1	grok	1	23.6k	4	libpng	1	467.6k	7	libsndfile	1	67.4k	18
libgit2	1	611.0k	3	libhevc	1	54.7k	11	libtiff	2	99.2k	14	libxml2	2	401.0k	17
libhtp	1	19.3k	1	libxml2	1	401.0k	3	lua	1	35.7k	4	openssl	6	913.0k	20
matio	1	35.0k	49	ndpi	1	42.9k	15	php	1	1.35M	16	poppler	3	241.0k	22
njs	1	132.0k	10	openh264	1	146.0k	22	sqlite3	1	1.1k	20				
php	2	1.35M	17	poppler	1	241.0k	17								
stb	1	93.6k	11	wireshark	1	5.27M	10								
zstd	1	110.0k	1												
Total: 16 B, 8.56M LoC, 175 Bugs								**Total: 18 B, 3.58M LoC, 138 Bugs**							
								Overall Total: 34 B, 12.14M, 313 Bugs							

fuzzers according to the current evaluation standards. FuzzBench primarily supports coverage-based benchmarking procedures, but also facilitates post-hoc bug-based evaluations. To facilitate a comparison with the results of Böhme et al. [7], who studied the agreement on outcome between a coverage-based and a bug-based evaluation of fuzzers in FuzzBench, we reuse their experimental setup and data.

Tools *T* (fuzzers). FuzzBench already integrates a diverse set of widely used fuzzers. The dataset produced by Böhme et al. [7] includes many AFL-based fuzzers (incl. AFL [51], and AFL++ [17], AFLFast [6], AFLSmart [37], FairFuzz [29], and MOPT [31]), two LibFuzzer-based fuzzers (LibFuzzer [42] and Entropic [5]), as well as Honggfuzz [45]. These fuzzers were used in the original study by Böhme et al. [7], represent the current state-of-the-art, and were also integrated into OSS-Fuzz [8]. From this dataset, we only exclude Eclipser [10] (which we also drop from our Magma experiments) as it does not contain trial data on three benchmarks.

Benchmarks *B*. FuzzBench [34] facilitates the swift integration of any of the hundreds of open source C/C++ programs that were previously added to the OSS-Fuzz [8] continuous fuzzing platform. All of these programs are deemed crucial to the security of the internet. To minimize experimenter bias, benchmarks and seed corpora are provided by the corresponding program maintainers.

For economic reasons, Böhme et al. [7] selected the benchmark set based on historical bug density, prioritizing programs that were known to contain a high number of previously identified bugs. To increase the statistical power of their analysis, they further selected only benchmarks where at least 30% of fuzzers discovered any bugs. Notably, in their experiments, they include a single benchmark for each of the programs, except for PHP, where they include two benchmarks. However, for simplicity, they elevate each of them to be an independent benchmark. As shown in Table 2, we include 16 benchmarks from the widely used open-source C/C++ programs found in the FuzzBench benchmarking suite, spanning multiple computing domains.

Bug identification (post-hoc). To mitigate survivorship and confirmation bias, Böhme et al. [7] use a *post-hoc analysis*. The bugs discovered by a fuzzer are identified in a semi-automated manner using AddressSanitizer [43] as a bug detector, the standard OSS-Fuzz deduplication strategy to cluster similar bug reports, and a manual deduplication to identify the unique set of bugs discovered by a fuzzer. Concretely, after running the fuzzing campaigns, they automatically reduced thousands of bug reports to 409 clusters and finally manually reduced this set to 235 unique bugs.

Computational infrastructure. The experiments are run within Ubuntu 16.04 Docker containers deployed on Google `n1-standard-1` instances. Each instance has one virtual CPU core, `3.75 GB` of RAM, and 30 GB of disk space. Böhme et al. [7] ran fuzzing campaigns for each fuzzer-benchmark combination for 23 hours, repeating each between 20 and 30 times, for a total of more than 11 CPU years. In our experiments, for each fuzzer × benchmark combination, we select the first 20 instances of the trials for analysis.

3.3 Magma: Fuzzers and Benchmarks

Magma [25] is primarily a bug-based benchmarking platform with a large number of known bugs in widely-used open source C/C++ programs. To maximize statistical power, Hazimeh et al. [25] forward-ported a large number of previously discovered bugs to a single, most recent version of the corresponding program.

Tools *T* (fuzzers). Magma supports a diverse set of widely-used fuzzers, including six of the nine fuzzers supported by FuzzBench (i.e., AFL [51], AFL++ [17], AFLFast [6], FairFuzz [29], MOPT [31], and Honggfuzz [45]). In addition to those six, Magma also supports an LLVM-based fuzzer that uses lightweight instrumentation called Instrim [26] and an LLVM-based symbolic-execution-driven whitebox fuzzer called SymCC[38]. Many Magma benchmarks are incompatible with in-process fuzzers (i.e., they cannot be linked against the benchmark). Consequently, we exclude LibFuzzer and Entropic from our Magma experiments.

Benchmarks *B*. We use Magma v1.2.0 [25], which integrates 21 benchmarks across nine open-source C/C++ programs widely used in security-critical settings. Magma provides a ground truth of 138 bugs that fuzzers are expected to find. The benchmark authors originally found these bugs in older versions of the programs and forward-ported them into the current versions. As shown in Table 2, we selected all 18/21 benchmarks that we were able to compile. Three benchmarks, namely, (`json`, `unserialize`, and `parser`) from the PHP library did not compile for most fuzzers. Moreover, consistent with the original Magma paper [25], SymCC fails to compile on exif, and we therefore omit results for this combination.

Bug identification (forward-porting). Unlike the post-hoc bug identification in FuzzBench, Magma uses manually inserted ”canaries” that automatically report when a fuzzer has reached or triggered the corresponding bug. A canary is implemented as an assertion that is violated when the bug-triggering condition is satisfied.

Computational infrastructure. The experiments are conducted on the Ubuntu 24.04.3 LTS 64-bit (Linux 6.14.0-24-generic) operating system running on a single HP ProLiant DL580 Gen9 machine equipped with a 4× Intel Xeon E7-8867 v4 processor running at 2.40 GHz (72 cores/144 threads) and 3.0 TiB RAM. The fuzzing campaigns for each fuzzer × benchmark combination were run for 23 hours and repeated 20 times, totaling *more than five (5) CPU years.*

Adding Support for Coverage Collection. The Magma experimental infrastructure does *not* support the collection of code coverage information out-of-the-box. We implemented a new feature to automatically record the current branch coverage achieved at regular intervals during a fuzzing campaign, and to generate an integrated report that combines Magma’s bug-finding results with our coverage results. In fact, the `llvm-cov` tool, part of the LLVM compiler infrastructure, did not support *branch* coverage until three versions *after* that, which is available in the default Magma Docker images (LLVM 9 vs LLVM 12). To enable branch coverage, we forked Magma to build the coverage harness with `LLVM 15` to match the `llvm-cov` version used by FuzzBench. We use this fork *only* for instrumentation and post-processing, so the fuzzing campaigns are unaffected. We provide this forked version of Magma and pipeline to the community for reproducibility (see section 9).

3.4 Measures of Fuzzer Performance

In a coverage-based benchmarking procedure, we measure the branch coverage achieved, whereas in a bug-based procedure, we measure the number of bugs found.

Code Coverage (#edges). Our measure of coverage is *branch coverage*, i.e., the number of control-flow edges (#edges) exercised over time during a fuzzing campaign (higher is better). Branch coverage subsumes statement coverage and is widely used as a proxy for code coverage due to its effectiveness [19, 20, 27]. We measure branch coverage directly on the buggy program to avoid the clean program assumption [9]. FuzzBench and Magma collect branch coverage using the LLVM compiler instrumentation and tooling (`llvm-cov`). Compiled with the appropriate compiler flags, the input queue of a fuzzing campaign is replayed, and per-input coverage is recorded with the corresponding time stamps. For Magma, we implemented the recording of branch coverage over time as a new feature.

Bug Finding (#bugs). Our measure of bug finding is the number of bugs found over time during a fuzzing campaign (higher is better). The total number of known bugs differs between benchmark programs: in Magma, this number is pre-determined, whereas in FuzzBench, it is unknown and determined post hoc through semi-automated analysis with a sophisticated de-duplication process. In Magma, we count the number of bugs "triggered".

Ranking fuzzers in terms of performance. We model the computation of the benchmarking outcome $R = outcome(P, T)$ exactly as in FuzzBench [7] and Magma [25]. Given a fuzz benchmark b, a set of fuzzers T, a performance measure m, a number of trials *#trials* (default 20), and a campaign length l, we compute a benchmark-specific ranking R_b by ranking each fuzzer $t \in T$ according to its mean performance on b under m, measured at the end of the length-l campaign and averaged over *#trials* trials, as follows.

$$R_b = \{r_t^b\}_{t\in T} = \text{Rank}\left(\left\langle \sum^{\#\text{trials}} \frac{m(t,b,l)}{\#\text{trials}} \right\rangle_{t\in T}\right) \tag{6}$$

Here, $m(t, b, l)$ denotes a random measurement of fuzzer t on benchmark b with respect to measure m at time l. The function Rank maps a sequence of values to an ordering s.t. larger values correspond to better performance (e.g., $\text{Rank}(\langle 300, 700, 400\rangle) = \langle 3, 1, 2\rangle$), breaking ties by random assignment within equivalence classes to ensure unique rankings. Given a benchmarking procedure $P = \langle B, m\rangle$, we define the overall benchmarking outcome as the ranking induced by the average rank across benchmarks. Since smaller ranks indicate better performance, we negate the average rank before applying Rank to ensure consistency:

$$outcome(P, T) = \text{Rank}\left(\left\langle -\frac{1}{|B|} \sum_{b\in B} r_t^b \right\rangle_{t\in T}\right). \tag{7}$$

Split-half and proxy split-half reliability. For each benchmark subset size $n \in \{1, \dots, \lfloor |B|/2 \rfloor\}$ and each evaluation metric $m \in \{\#edges, \#bugs\}$, we estimate both *split-half reliability* (Equation 3) and *proxy split-half reliability* (Equation 5) by sampling up to 10,000 unique disjoint benchmark subset pairs $\langle B_1, B_2\rangle$ without replacement where $|B_1| = |B_2| = n$ and $B_1 \cap B_2 = \emptyset$ and reporting the mean correlation across samples. For efficiency, we avoid duplicated pairs (i.e., if $\langle B_1, B_2\rangle$ is sampled, then $\langle B_2, B_1\rangle$ is not). Our Magma experiment contains 18 benchmarks ($n \le 9$), while FuzzBench contains 16 benchmarks ($n \le 8$).

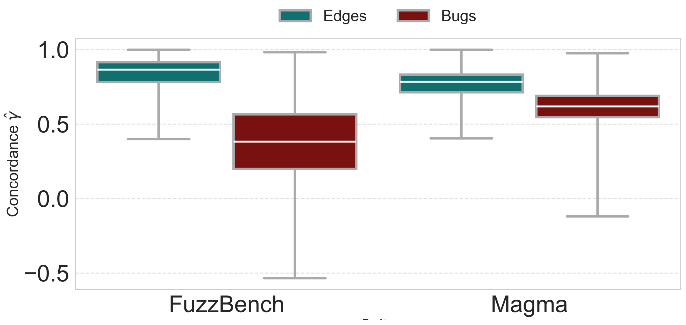
FuzzBench ($|B| = 16$)

m	$\hat{\gamma}$	Interpretation
#edges	0.843	ST
#bugs	0.373	WK

Magma ($|B| = 18$)

m	$\hat{\gamma}$	Interpretation
#edges	0.782	ST
#bugs	0.603	MD

(a) Mean split-half reliability.



(b) Split-half reliability distributed over 10,000 randomly sampled benchmark subset pairs of size $\frac{|B|}{2}$, where $|B|_{FB} = 16$ and $|B|_{MG} = 18$.

Fig. 1. Concordance of bug- and coverage-based benchmarking for Magma and FuzzBench.

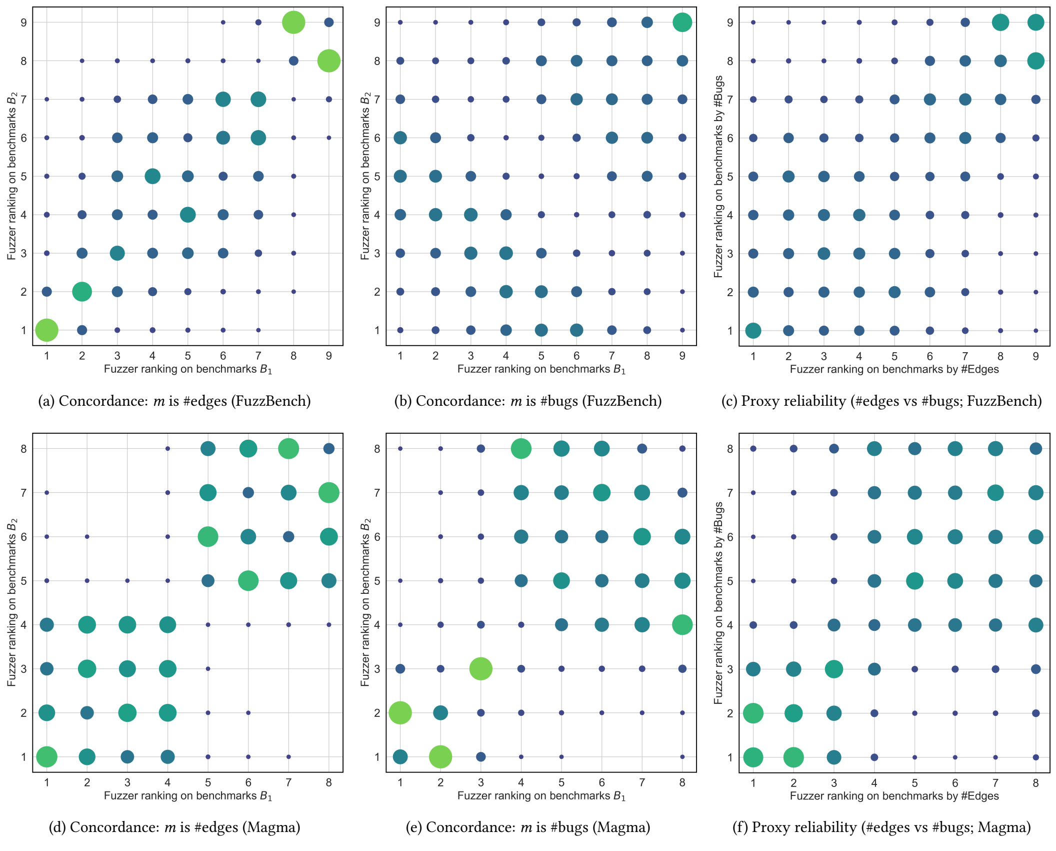


Fig. 2. Scatter plots of fuzzer rankings on metrics X vs. Y over 20 trials of 23 hours ($X, Y \in$ {#edges, #bugs}). Circle size and color indicate frequency; colors span deep blue (1) to bright green (10000). For concordance plots (a, b, d, e), rank pairs are treated as unordered (e.g., $(1, 3)$ and $(3, 1)$), and their frequencies are averaged.

4 Experimental Results

RQ.1 Concordance of Bug-based versus Coverage-based Benchmarking

We study the concordance of a bug-based versus a coverage-based fuzzer benchmarking procedure. In other words, we empirically measure how reliable the outcome of a bug-based evaluation of fuzzer performance is in comparison to that of a coverage-based evaluation.

Presentation. Figure 1.a shows the concordance between fuzzer rankings induced by branch coverage (#edges) and bug discovery (#bugs). Figure 1.b shows the distribution of split-half reliability over 10,000 random, disjoint pairs of benchmark subsets of size $\frac{|B|}{2}$, computed separately for each performance measure on FuzzBench and Magma. Finally, Figure 2 depicts scatterplots of fuzzer rankings across two disjoint benchmark subsets, using #edges (Fig. 2.a+d) and #bugs (Fig. 2.b+e), for 10,000 random pairs (subset size 8 for FuzzBench and 9 for Magma).

Results. As shown in Figure 1a, coverage-based benchmarking achieves higher concordance (0.78 on Magma, 0.84 on FuzzBench) than bug-based benchmarking (0.60 on Magma, 0.37 on FuzzBench). While coverage-based concordance is consistently *strong*, bug-based concordance is at best *moderate*. This suggests that coverage-based benchmarking provides more reliable and consistent fuzzer rankings across benchmark subsets.

Notably, bug-based concordance differs sharply between Magma (0.60) and FuzzBench (0.37). We hypothesize that this gap arises because Magma was explicitly designed as a ground-truth bug benchmarking suite with over a hundred forward-ported bugs, whereas FuzzBench uses bugs that organically exist and are thus much more sparsely distributed. We elaborate on these results below.

Coverage-based benchmarking (#edges). As shown in Figure 1, coverage-based benchmarking exhibits *strong* (ST) concordance on both Magma and FuzzBench. In other words, if we ranked fuzzers by branch coverage on two random benchmark subsets of equal size, the rankings would strongly agree. The green box plots in Figure 1b further show that the variance of split-half reliability across 10,000 randomly sampled disjoint pairs is low compared to bug-based benchmarking.

This concordance is most clearly illustrated in Figure 2. For FuzzBench (Fig. 2.a), points closely aligned with the diagonal indicate strong agreement between ranks. For Magma (Fig. 2.a), two distinct clusters emerge: fuzzers consistently ranked in the Top-4 (AFL++ [17], MOPT [31], Honggfuzz [45], and SymCC [38]) and those consistently in the Bottom-4. A Top-4 fuzzer on one benchmark subset rarely falls below fifth on another set. For more technical details on the performance of these fuzzers, we refer to Hazimeh et al. [25].

Bug-based benchmarking (#bugs). As shown in Figure 1, bug-based benchmarking yields markedly different concordance distributions on Magma and FuzzBench. On Magma, concordance is *moderate* (MD, 0.60). The red box plots in Figure 1b show relatively low variance, with the median half above the moderate (MD) threshold. Only the lower quartile reaches into disagreement ($\gamma < 0$; as low as NWK). Figure 2.e exhibits clustering similar to Magma's coverage-based evaluation.

On FuzzBench, by contrast, concordance is only *weak* (WK, 0.37) as shown in Figure 1.a. The ranking of fuzzers on FuzzBench in terms of the number of bugs found may not always agree very well across two equi-sized random benchmark subsets. Figure 1.a shows a wide spread of split-half reliability values. The lowest quartile even reaches into moderate disagreement ($\gamma \leq -0.4$, NMD). This spread is confirmed in Figure 2.b, which reveals a subtle diagonal valley starts from (1,6) to (6,1), indicating that a fuzzer ranked first on one subset often falls to fourth through sixth on the other. A manual inspection of the FuzzBench data suggests that this pattern arises from the wide variation in an individual fuzzer's performance across different benchmarks. A fuzzer may excel on one benchmark but perform poorly on others. As a result, whether a benchmark that favors a

Table 3. Concordance of coverage- and bug-based benchmarking procedures, measured via split-half reliability, and inter-measure agreement on ranking between coverage- and bug-based fuzzer performance, both evaluated on benchmark subsets of size $n = \lfloor |B|/2 \rfloor$.

Test		**FuzzBench**								**Magma**								
	n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Split-half	#edges	0.309	0.438	0.572	0.640	0.718	0.769	0.809	0.843	0.346	0.474	0.601	0.659	0.709	0.734	0.754	0.767	0.781
	#bugs	0.067	0.103	0.154	0.220	0.258	0.302	0.339	0.373	0.181	0.274	0.368	0.442	0.500	0.545	0.578	0.599	0.603
Proxy (#e vs #b)		0.162	0.245	0.347	0.421	0.483	0.541	0.590	0.634	0.187	0.328	0.414	0.475	0.518	0.552	0.578	0.599	0.613

given fuzzer falls into the first half or the second half can substantially change that fuzzer's rank, leading to large differences between the two half-set rankings.

RQ.1*: While the concordance of coverage-based benchmarking is strong in both Magma and FuzzBench, the concordance of bug-based benchmarking is only weak to moderate. In other words, if we took two random benchmark subsets of equal size and ranked fuzzers by bug finding, the resulting rankings would often differ. By contrast, if we ranked the same fuzzers by coverage, we would expect the rankings to be almost identical. Across benchmarks, bug-based concordance is notably higher on Magma (0.60), which includes a wide set of forward-ported bugs, than on FuzzBench (0.37).*

RQ.2 Concordance as a Function of Benchmarking Suite Size

We study how the concordance of coverage- and bug-based benchmarking procedures changes as the size of the benchmarking suite increases.

Presentation. Table 3 reports concordance (split-half reliability) for benchmarking suite of size $2n$, i.e., the agreement between rankings obtained from two disjoint subsets of size n. 'Split-half' rows correspond to rankings based on branch coverage (#edges) and bug discovery (#bugs) Figure 3 complements this by visualizing how concordance changes with benchmark subset size: the dotted green line for #edges and the solid red line for #bugs.

Results. Concordance generally increases with the number of benchmarks for both performance measures and both benchmarking suites. In comparison, coverage-based benchmarking shows a noticeably faster rise in concordance than bug-based benchmarking. For both Magma and FuzzBench, split-half reliability of coverage-based benchmarking exceeds the *strong* threshold ($\gamma > 0.7$) once the benchmarking suite size reaches $2n \geq 10$ (i.e., $n \geq 5$). In contrast, bug-based benchmarking on FuzzBench only attains the *weak* threshold ($\gamma \geq 0.1$) at $2n = 2$ ($n = 1$) and remains at that level up to the maximum size of $2n = 16$ ($n = 8$). On Magma, bug-based concordance reaches the *moderate* threshold ($\gamma \geq 0.4$) relatively early at $2n = 8$ ($n = 4$). Still, the rate of increase diminishes as the set grows, and it stays below the *strong* threshold ($\gamma \geq 0.7$) until $2n = 18$ ($n = 9$).

Single benchmark (n = 1). As a special case, we examine the agreement in benchmarking outcomes when the same set of fuzzers is evaluated on one benchmark and then on another. On FuzzBench, bug-based benchmarking shows only a *negligible agreement* (NG, 0.07): On FuzzBench, the ranking of fuzzers in terms of the number of bugs found in one benchmark is effectively random with respect to the ranking on another benchmark. On Magma, this agreement is *weak* (WK, 0.18), though barely above negligible. This suggests that using a single benchmark ($n = 1$) renders bug-based benchmarking largely unreliable.

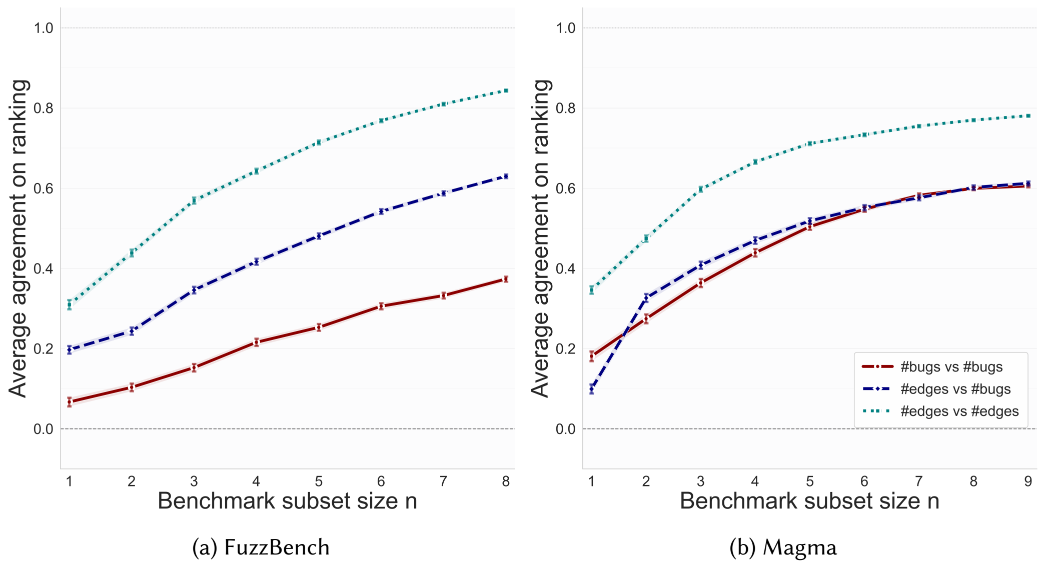


Fig. 3. Concordance of coverage-based (green) and bug-based (red) benchmarking procedures, measured by split-half reliability, and inter-measure agreement between coverage-based and bug-based fuzzer performance rankings (blue). All quantities are average agreement on ranking (y-axis), evaluated on benchmark subsets of size $n = \lfloor |B|/2 \rfloor$ (x-axis). Increasing n increases ranking stability by averaging over more benchmarks.

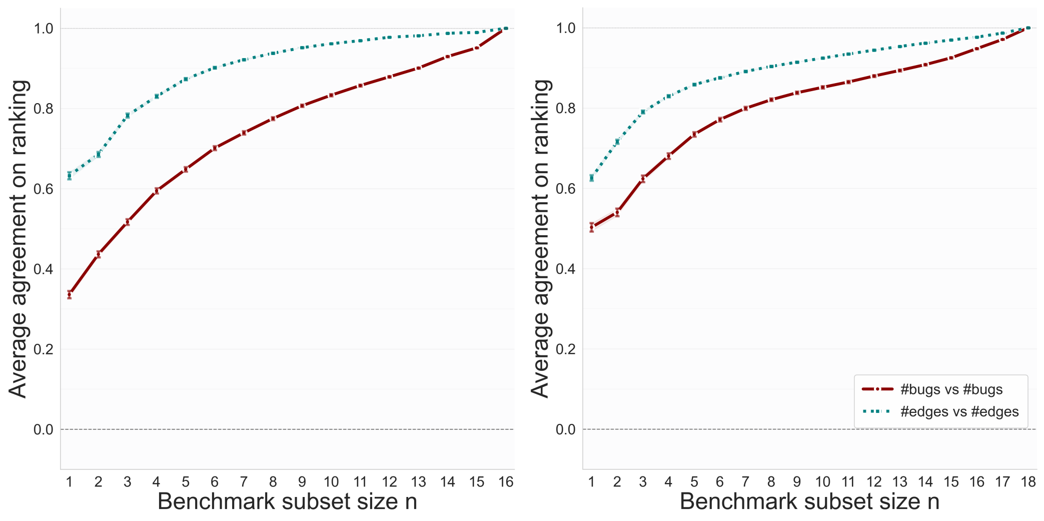
On Magma, bug-based benchmarking only reaches a *moderate agreement* (MD) after it reaches benchmark subset size $n = 4$, whereas on FuzzBench it never rises above weak levels. In contrast, even at $n = 1$, coverage-based benchmarking already achieves a *weak agreement* (WK, 0.35 on Magma and 0.31 on FuzzBench), which quickly rises to *moderate* (MD) at $n = 2$ and *strong* (ST) at $n = 5$. Taken together, these results highlight the superior reliability of coverage-based benchmarking, even with minimal benchmark sets.

***RQ.2**: Concordance increases with benchmarking suite size for both coverage- and bug-based benchmarking. However, coverage-based benchmarking achieves strong concordance with as few as five benchmarks, while bug-based benchmarking remains weakly concordant on FuzzBench and only moderately concordant on Magma, even with largest possible benchmark subset pairs. Notably, with a single Benchmark ($n = 1$), coverage-based benchmarking already attains weak concordance, i.e., there is some agreement between the rankings of fuzzers on two benchmarks in terms of coverage, whereas bug-based benchmarking is largely unreliable.*

RQ.3 Split-half Reliability versus Proxy Split-half Reliability

We investigate the comparison between the split-half reliability (as a measure of concordance) of bug-based benchmarking and the proxy split-half reliability of coverage with respect to bugs. If the proxy reliability of #edges with respect #bugs exceeds the mean split-half reliability of #bugs-based benchmarking, then coverage-based benchmarking predicts the outcome of a bug-based evaluation more reliably than bug-based benchmarking on itself.

Presentation. Figure 2.c+f shows scatter plots of proxy reliability, comparing fuzzer rankings by #edges on one half of the benchmarks with their rankings by #bugs on the other half. The last row



(a) FuzzBench (b) Magma

Fig. 4. Agreement on ranking between a benchmarking procedure that uses the entire benchmarking suite and one that uses a benchmark subset of size *n* for coverage- and bug-based benchmarking.

of Table 3 ('Proxy (#b vs #e)') reports how proxy reliability changes with benchmark subset size. Finally, the dashed blue line in Figure 3 visualizes how proxy reliability evolves as the benchmark subset size increases.

Results. For FuzzBench, the concordance of bug-based benchmarking is consistently worse than the proxy split-half reliability of coverage- with respect to bug-based benchmarking. In other words, evaluating the fuzzers in terms of #edges offers a more reliable prediction of the outcome of an evaluation in terms of #bugs than an independent evaluation in terms of the #bugs itself. Proxy split-half reliability relations increase monotonically with benchmark subset size, in line with the increase in concordance. On FuzzBench, when the set size reaches $n = 8$, the proxy reliability achieves *moderate agreement* (MD, 0.63), which is substantially higher than the bug-based split-half reliability (0.37). This trend is persistent across all set sizes: in most cases, the proxy reliability is nearly double that of the bug-based reliability.

This observation can be interpreted in light of the example discussed earlier comparing study time and exam scores (Section 2.1). Bug-based benchmarking resembles exam scores with limited discriminative power: bugs are sparse, their locations are unknown a priori, and triggering them often depends on highly specific inputs, making outcomes sensitive to randomness. As a result, performance on one benchmark subset does not necessarily translate into consistent performance on another, leading to low split-half reliability. In contrast, coverage-based measures are analogous to study time, reflecting a fuzzer's sustained exploration effort rather than rare events. Consequently, coverage-based rankings tend to be more stable across benchmark subsets and better predict bug-based outcomes on unseen benchmarks than an independent bug-based evaluation itself.

For Magma, the concordance of bug-based benchmarking is at least comparable to the proxy split-half reliability of coverage- with respect to bug-based benchmarking. Bug-based benchmarking already achieves a *moderate* level of split-half and proxy reliability (MD, 0.60) at half size ($n = 9 = |B|/2$). The interpretation levels are generally identical across set sizes, except for $n = 3$, where bug-based reliability is classified as *weak* (WK, 0.37) while the proxy reliability already reaches a

Table 4. For each similarity threshold θ, we report the smallest $|B'|$ that attains that agreement (top) and the corresponding tree-years (bottom), computed as $(|B'|/|B|) \times TY$, where $|B|_{FB} = 16$, $|B|_{MG} = 18$, and $TY_{FB} \approx TY_{MG} \approx 6$ tree-years.

Benchmarking Procedure	**FuzzBench**					**Magma**				
	$\theta = 0.5$	0.7	0.8	0.9	0.95	$\theta = 0.5$	0.7	0.8	0.9	0.95
#edges	1	3	4	6	9	1	2	4	8	13
#bugs	3	6	9	13	15	1	5	8	14	17

moderate level (MD, 0.41). This outcome aligns with expectations: since bug-based benchmarking on Magma already exhibits moderate concordance, using edge coverage as a proxy does not yield substantial additional gains in predicting the outcome of bug-based evaluation. In other words, the Magma bug exam provides a more reliable and discriminative signal than the FuzzBench bug exam. However, it does offer marginal improvements in the smaller subsets.

***RQ.3**: Surprisingly, the outcome of a coverage-based evaluation is at least as reliable (and for FuzzBench much more reliable) as a predictor of the outcome of a bug-based evaluation than an independent bug-based evaluation with a different, equi-sized benchmark subset. On FuzzBench, where bug-based concordance is weak, the proxy reliability is nearly twice as reliable as the baseline, revealing that coverage-based benchmarking can be a more consistent predictor of bug-based outcomes than bug-based benchmarking itself. On Magma, however, where bug-based concordance is already moderate, a coverage-based evaluation brings only marginal gains, underscoring that the striking advantage of coverage-based proxies emerges most clearly when bug-based benchmarking performs poorly.*

5 Concordance as Measure of Benchmarking Efficiency

To explore an application of concordance, we experimentally investigate whether a *benchmarking procedure with greater concordance is also more resource efficient.* This is an important consideration for *green fuzzer benchmarking* [35].

Carbon emissions of fuzzer benchmarking. We estimate that the experiments for Magma and FuzzBench each emitted as much carbon dioxide as *six trees can absorb in one year*, i.e., six tree-years. To maintain focus on the main hypothesis, we postpone the concrete modelling of these carbon emissions to Appendix A. FuzzBench ran nine fuzzers on 16 benchmarks for 23 hours, repeating these experiments 20 times on `n1-standard-1` Google Cloud VM instances in the `us-west1` region. For Magma, we assume the same configuration but running 8 fuzzers on 18 benchmarks for 23 hours. This means that one CPU year of fuzzer benchmarking corresponds roughly to $\frac{3}{4}$ tree years.

We are the first to model and study the carbon emissions of fuzzer benchmarking. This allows us to quantify the savings of green fuzzer benchmarking [2, 35] and to study the reduction of carbon emissions as a trade-off w.r.t. the reliability of the benchmarking outcome. Indeed, the environmental cost of cloud computing can be substantial and is of general interest [3, 36, 40].

5.1 Green Fuzzer Benchmarking: Reducing Benchmarking Suite Size

The benchmark set (i.e., the benchmarking suite) of a procedure with greater concordance can be more substantially reduced while maintaining a similar outcome. Specifically, given the similarity threshold θ, benchmarking procedures $P_1 = \langle B, m_1 \rangle$ and $P_2 = \langle B, m_2 \rangle$, as well as fuzzers T, let $B_1 \subseteq B$

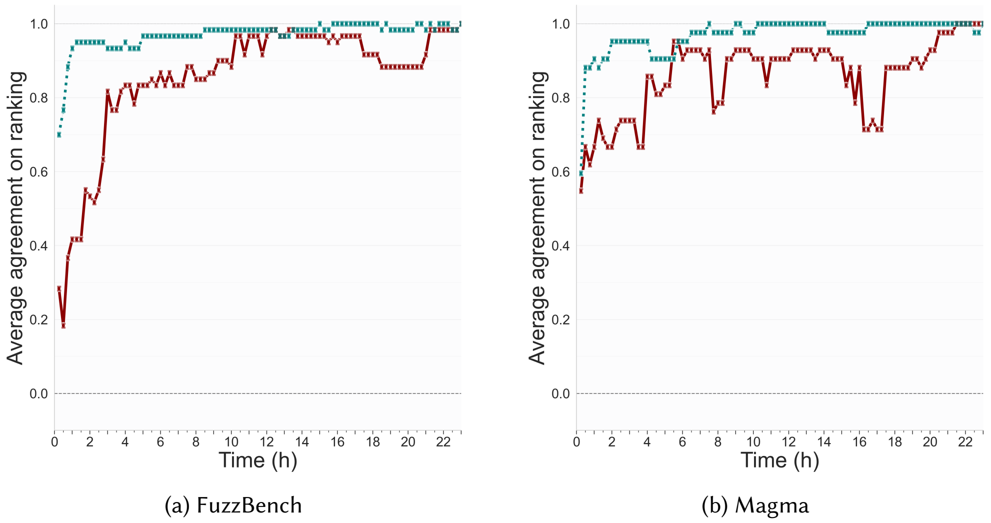


Fig. 5. Agreement on ranking y between a fuzzing campaign of length x and one of length 23 hours for coverage- and bug-based benchmarking.

be the smallest subset of b s.t. $\theta \leq \mathit{agreement}(\mathit{outcome}(\langle B_1, m_1 \rangle, T), \mathit{outcome}(\langle B, m_1 \rangle, T))$ and let $B_2 \subseteq B$ be the smallest subset of b s.t. $\theta \leq \mathit{agreement}(\mathit{outcome}(\langle B_2, m_2 \rangle, T), \mathit{outcome}(\langle B, m_2 \rangle, T))$. We study if the following is true: If $\gamma(P_1, T) > \gamma(P_2, T)$, then $|B_1| \leq |B_2|$.

Figure 4 shows how similar the outcome of a benchmarking procedure is if the benchmark set was reduced to n benchmarks—for bug-based and coverage-based benchmarking, for FuzzBench and Magma. Table 4 illustrates the savings in carbon emissions as a tradeoff against the reliability of the benchmarking outcome. Given a similarity threshold θ, the table shows how many benchmarks can be removed while preserving a similar outcome for coverage- and bug-based benchmarking.

Observation: *For both Magma and FuzzBench, we find that the benchmark set of coverage-based benchmarking—which exhibits higher concordance—can be reduced more substantially while maintaining a similar benchmarking outcome as that of bug-based benchmarking.*

For instance, as we can see in Table 4, using half of the benchmarks (i.e., fuzz drivers) would be sufficient for coverage-based benchmarking to achieve at least a *very strong agreement* on outcome ($\theta = 0.9$) compared to running the procedure with the whole benchmarking suite. Yet, three-quarters of the benchmark set is barely enough for FuzzBench to achieve the same similarity threshold for bug-based benchmarking. The reduction in carbon emissions is more than twice for coverage-based compared to bug-based benchmarking. For coverage-based benchmarking, even if we used only three benchmarks ($n = 3$), we would achieve a *strong agreement* on outcome if we used the entire benchmark set, which is five to six times larger, while for bug-based benchmarking, such a small benchmark set would only achieve a *moderate agreement*.

5.2 Green Fuzzer Benchmarking: Reducing Campaign Length

We study whether the campaign length of a procedure with greater concordance can be more substantially reduced while maintaining a similar benchmarking outcome. Specifically, given a similarity threshold θ, two procedures P_1 and P_2, fuzzers T and campaign length l, let $\mathit{outcome}(P, T, l)$ be

Table 5. For each similarity threshold θ, we show the earliest time to reach it in minutes (top), and the corresponding tree-years (bottom), computed as $(|B'|/|B|) \times TY$, where $|B|_{FB} = 16$, $|B|_{MG} = 18$, and $TY_{FB} \approx TY_{MG} \approx 6$ tree-years.

Benchmarking	**FuzzBench**					**Magma**				
Procedure	$\theta = 0.5$	0.7	0.8	0.9	0.95	$\theta = 0.5$	0.7	0.8	0.9	0.95
#edges	15 ○○○ ○○○	15 ○○○ ○○○	45 ○○○ ○○○	60 ◐○○ ○○○	300 ●◐○ ○○○	15 ○○○ ○○○	30 ○○○ ○○○	30 ○○○ ○○○	90 ◐○○ ○○○	345 ●◐○ ○○○
#bugs	105 ◐○○ ○○○	180 ◐○○ ○○○	180 ◐○○ ○○○	1260 ●●● ●●◐	1275 ●●● ●●◐	15 ○○○ ○○○	240 ●○○ ○○○	240 ●○○ ○○○	1185 ●●● ●●○	1230 ●●● ●●◐

the outcome of P on T at time l and let ϕ_θ s.t. $\theta = agreement(outcome(P_1, T, \phi_\theta l), outcome(P_1, T, l))$. In other words, the campaign length of P_1 can be reduced by a factor of ϕ_θ while maintaining a similar benchmark outcome (as determined by the similarity threshold θ). We study if the following is true: If $\gamma(P_1, T) > \gamma(P_2, T)$ at time l, then $agreement(outcome(P_2, T, \Phi_\theta l), outcome(P_2, T, l)) < \theta$.

Observation: *For both Magma and FuzzBench, we find that the campaign length of coverage-based benchmarking—which exhibits higher concordance—can be reduced more substantially while maintaining a similar benchmarking outcome as that of bug-based benchmarking.*

As we can see in Figure 5, the agreement on the final ranking does vary quite substantially over campaign length. Yet, the agreement with the final ranking of coverage-based benchmarking is consistently higher than that of bug-based benchmarking. Clearly, there are more savings to be expected for coverage-based benchmarking. Table 5 shows that running a coverage-based campaign for 1 hour (or 1.5 hours) instead of 23 hours achieves benchmark outcomes that strongly agree, but with carbon emissions of less than 3 tree months on FuzzBench (or 4.5 tree months on Magma). Independently, Ounjai et al. [35] observe an agreement of 0.83 between a fuzzing campaign of length 15 minutes (and substantially more benchmarks) and one of 23 hours in FuzzBench for coverage-based benchmarking, which approximately aligns with our experiments.

6 Threats to Validity

We recognize several potential validity threats, which we categorize as internal, external, construct, and conclusion validity. Each category outlines possible limitations and our mitigation strategies.

Internal Validity. The internal validity of our study could be subject to selection biases introduced by Hazimeh et al. [25] and Böhme et al. [7] choice of programs, benchmarks, and fuzzers in their respective studies. We mitigate these threats by analyzing random trials and benchmark subset pairs, reducing the influence of randomness in fuzzing campaigns. We further analyze filtered data from both FuzzBench and Magma, limiting bias from the idiosyncrasies of any single dataset. Because FuzzBench draws its programs from OSS-Fuzz, it emphasizes bug-prone software and may suffer from survivorship and selection biases [7]. However, since our study evaluates concordance among benchmarking metrics rather than ranking fuzzers, these biases have limited impact on our conclusions. Moreover, FuzzBench and Magma are popular and widely used, independently developed fuzzer benchmarking platforms.

External Validity. The external validity, which pertains to the generalizability of our study to other contexts, is affected by both the bug-detection approaches in Magma [25] and FuzzBench [7, 33] and the diversity of benchmarks and fuzzers. Magma forward-ports bugs to the latest program versions, while FuzzBench relies on symbolized stack frames from crash reports [25, 33]. While this already covers many bugs that fuzzer practitioners care about, it does not encompass all classes of bugs that do not result in crashes. Additionally, both datasets are constructed using C/C++ programs and (mostly) gray-box fuzzers, which limits their applicability to other programming languages and bug-detection methods. Moreover, our filtering for experimental consistency may bias results toward a few particularly bug-prone benchmarks. To partially mitigate this threat, we evaluate concordance in benchmarking results as the benchmark subset size increases (see section 4). A broader evaluation will require more diverse programs, bugs, and fuzzing metrics.

Conclusion Validity. In this work, we treat fuzz harnesses as the unit of benchmarking. A natural concern is that programs with more harnesses could exert disproportionate influence on the benchmarking outcome. Mean split-half reliability mitigates this effect by construction. However, it is not robust to artificial ties, for example, when multiple harnesses within a program tend to detect the same bugs, thereby increasing the frequency of equal ranks. To address this concern, we recomputed all analyses at the program level by aggregating each fuzzer's performance across the harnesses of each program. The interpretation of the results remained essentially unchanged, except for one benchmarking suite procedure combination.

7 Related Work

Böhme et al. [7] found that while a strong correlation exists between a fuzzer's ability to discover bugs and its code coverage, rankings based on coverage *do not strongly agree* with those based on bug-finding effectiveness. This discrepancy motivates our study, which analyzes the concordance of fuzzer benchmarking procedures. Unlike their work, which examines the agreement between different benchmarking metrics, we focus on the concordance of benchmarking procedures, and specifically, the self-agreement between bug-based and coverage-based metrics. Our work reflects the reliability of benchmarking procedures, offering a more nuanced understanding of the trustworthiness of bug-based and coverage-based metrics.

Zeller and Just [52] caution that incentivizing researchers to focus on success on a given benchmark ranking risks missing valuable insight and can lead to fragile conclusions stemming from results that overfit to the benchmark and may not generalize. They also highlight that tools contributing to new code coverage gains remain very much relevant even if these gains do not lead to finding new bugs. Our work introduces a framework for measuring the stability of benchmarking procedures. Across suites, bug-based rankings are noisy and sensitive to the choice of benchmark set and the bug selection approach (e.g., ad-hoc vs. forward porting). However, coverage-based rankings are substantially more stable, enabling repeatable comparative insight.

In recent work, researchers frequently use code coverage as a proxy for assessing the bug-finding capabilities of test suites, based on the assumption that *uncovered code cannot reveal bugs* [7]. However, Li et al. [30] caution against relying on a single metric for fuzzer evaluation, advocating for a multidimensional approach that captures diverse performance characteristics. They argue that variations in instrumentation and crash analysis can introduce bias. Code coverage—typically measured by exercised program branches often correlates with bug-finding metrics like bug count or time to first bug, but its reliability as a predictor remains under scrutiny. Gopinath et al. [24] find strong correlations between coverage and bug discovery for developer-provided test suites and moderate-to-strong correlations for auto-generated suites, despite low coverage levels. Similarly, Gligoric et al. [20] report strong coverage-bug discovery correlations. In contrast, Wei et al. [49] highlight that over 50% of bugs emerge in the late stages of fuzzing, where coverage gains are minimal, suggesting a weaker correlation between coverage and fault discovery.

Beyond coverage-based concerns, recent work critiques the reliability of fuzzer evaluation metrics, highlighting biases in bug-based measures and statistical shortcomings in benchmarking procedures. Gavrilov et al. [18] argue that bug-based metrics suffer from ambiguities in defining "bugs" and challenges in mapping inputs to bug discovery, proposing an evaluation framework based on behavioral changes over time. Schloegel et al. [39] emphasize statistical rigor in fuzzing evaluations, warning that insufficient trial repetitions and weak statistical analyses undermine reproducibility. They call for stricter evaluation guidelines to improve research reliability. Unlike prior work, our study examines both intra-metric agreement (internal consistency) and inter-metric agreement across coverage- and bug-based metrics to assess their predictive reliability.

Limitations in current benchmarks are further underscored by Zhang et al. [53], who argue that existing benchmarks often fail to capture the complexity of real-world bugs due to challenges in translating intricate vulnerabilities into standardized tests. Hazimeh et al. critique fuzzer evaluation metrics such as crash count, noting inaccuracies stemming from imperfect deduplication. They propose comparing fuzzers based on real-world bugs found, despite the lack of a standardized "bug" definition. Their work introduces a benchmarking suite that incorporates real bugs into actual software, aiming to standardize fuzzer evaluation across diverse benchmarks [25]. To our knowledge, this is the first rigorous evaluation of *concordance* in fuzzer benchmarking, assessing the consistency of these metrics across different benchmark sets.

8 Conclusion

In this work, we establish concordance as an important property for assessing the reliability of a benchmarking procedure. A procedure with low concordance yields fairly unreliable outcomes, characterized by low internal consistency. For instance, we found that a bug-based evaluation in FuzzBench has a weak concordance. So weak in fact that the outcome of a coverage-based evaluation agrees more with the outcome of a bug-based evaluation than with the bug-based evaluation itself.

Our analysis of FuzzBench and Magma results suggests that if we require a bug-based benchmarking evaluation to have a high concordance, then its outcome aligns very well with code coverage evaluation. In some sense, coverage-based benchmarking produces outcomes that are at least somewhat representative of a bug-based evaluation with high concordance.

9 Data Availability and Reproducibility

For transparency and reproducibility, we provide our scripts, which contain the full data analysis script along with all generated tables and figures. This resource is available at:

https://github.com/ardier/in_bugs_we_trust/

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and for helping us improve this paper. This research is partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is supported by ERC grant (Project AT_SCALE, 101179366). It is also partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

A Modelling Energy Consumption of Fuzzing Campaigns

We use the modeling of carbon emissions from cloud resources by Lannelongue et al. [28], which is widely used in academia. Böhme et al. [7] conduct 23-hour trials on a `n1-standard-1` VM (1 vCPU $\approx$ one core, 3.75 GiB RAM) [23]. Among the five Intel Xeon models supporting this instance, we use the E5-2696v2's specifications, as it is a representative mid-range CPU.

The energy consumption E for the FuzzBench experiments is estimated as follows:

$$E = t \cdot (n_c \cdot P_c \cdot u_c + n_m \cdot P_m) \cdot \text{PUE} \tag{8}$$

where $t = 23$ hours, $n_c = 2880 = 16$ benchmarks $\cdot$ 9 fuzzers $\cdot$ 20 trials, per-core draw $P_c = 10$W [12], $u_c = 1$ (we assume full per-core utilization because fuzzers saturate CPUs), $n_m = 10800 = n_c \cdot 3.75$ GiB of RAM, $P_m = 0.375$ W per GiB [14, 28], and PUE is dependent on the cloud service provider. For `us-west1`, a low-carbon region [22], Google reports a trailing twelve-month average PUE of 1.1 for this region [21]. Finally, we get $E = 831.11$ kWh.

According to World Resources Institute [50], the location-based carbon emissions are given by:

$$C = E \times \text{CI}. \tag{9}$$

where CI is the carbon intensity (in KgCO$_2$eq/kWh) for the specific cloud region [28] and E is the energy consumption (in kWh) of the fuzzing experiments. The `us-west1` region has a grid carbon intensity of 79 gCO$_2$e/kWh, i.e., less than half that of the second-lowest North American region (`us-west2` at 169 gCO$_2$e/kWh) [22]. Hence, the carbon footprint C (in KgCO$_2$e) is computed as $C = 65.658$ KgCO2eq.

A common way to communicate carbon emissions is as the number of *tree-years* required to sequester the emitted CO$_2$e. Let $S_T = 11$ kg CO$_2$e tree^{-1} yr^{-1} denote the amount a mature tree sequesters in one year ($\approx \frac{30\text{ g}}{\text{day}}$), following Akbari [1]. Thus, the required tree-years to sequester the emissions TY is $TY = \frac{C_{FB}}{S_T} = \frac{65.658}{11} \approx 6$ tree-years.

Assuming the same configuration for Magma, we estimate $E_{\text{MG}} = 825.33$ kWh, $C_{\text{MG}} = E_{\text{MG}} \times \text{CI} = 825.33 \times 0.079 = 65.20$ kg CO$_2$e, and $TY_{\text{MG}} = C_{\text{MG}}/S_T = 65.20/11 = 5.93$ tree-years.

References

[1] Hashem Akbari. 2002. Shade trees reduce building energy use and CO2 emissions from power plants. *Environmental pollution* 116 (2002), S119–S126.

[2] Seyed Behnam Andarzian, Cristian Daniele, and Erik Poll. 2023. Green-Fuzz: Efficient fuzzing for network protocol implementations. In *International Symposium on Foundations and Practice of Security*. Springer, 253–268.

[3] Hanan Awwad, Changyuan Lin, Rabab Ward, and Mohammad Shahrad. 2025. Estimating the Carbon Footprint of Serverless Functions on a Public Cloud Platform. In *Proceedings of the 3rd Workshop on SErverless Systems, Applications and MEthodologies*. 12–20.

[4] 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.

[5] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[7] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1621–1633. doi:10.1145/3510003.3510230

[8] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2025-03-04.

[9] 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)*. IEEE, IEEE, Buenos Aires, Argentina, 597–608.

[10] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.

[11] Jose M Cortina. 1993. What is coefficient alpha? An examination of theory and applications. *Journal of applied psychology* 78, 1 (1993), 98.

[12] CPU Benchmark. n.d.. Intel Xeon E5-2696V2 Processor 2.5 GHz - CPU Benchmark. https://www.cpubenchmark.net. Accessed: 2025-09-04.

[13] Lee J Cronbach. 1951. Coefficient alpha and the internal structure of tests. *psychometrika* 16, 3 (1951), 297–334.

[14] Crucial. [n. d.]. How much power does memory use? https://www.crucial.com/support/articles-faq-memory/how-much-power-does-memory-use. [Accessed 05-09-2025].
[15] Haroon Elahi and Guojun Wang. 2024. Forward-porting and its limitations in fuzzer evaluation. *Information Sciences* 662 (2024), 120142. doi:10.1016/j.ins.2024.120142
[16] Leonard S Feldt. 1969. A test of the hypothesis that cronbach's alpha or kuder-richardson coefficent twenty is the same for two tests. *Psychometrika* 34, 3 (1969), 363–373.
[17] 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, Virtual Event, 12 pages. https://www.usenix.org/conference/woot20/presentation/fioraldi
[18] Miroslav Gavrilov, Kyle Dewey, Alex Groce, Davina Zamanzadeh, and Ben Hardekopf. 2020. A practical, principled measure of fuzzer appeal: A preliminary study. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 510–517.
[19] Gregory Gay. 2017. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering*. Springer, 65–82.
[20] 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 Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 1–33.
[21] Google. n.d.. Power usage effectiveness. https://datacenters.google/efficiency/. Accessed: 2025-09-04.
[22] Google Cloud. n.d.. Carbon free energy for Google Cloud regions. https://cloud.google.com/sustainability/region-carbon. Accessed: 2025-09-04.
[23] Google Cloud. n.d.. CPU platforms - Compute Engine Documentation. https://cloud.google.com/compute/docs/cpu-platforms. Accessed: 2025-09-04.
[24] 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*. 72–82.
[25] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. doi:10.1145/3428334
[26] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, Vol. 40. NDSS, San Diego, CA, USA, 7 pages.
[27] 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*. 2123–2138.
[28] Loïc Lannelongue, Jason Grealey, and Michael Inouye. 2021. Green algorithms: quantifying the carbon footprint of computation. *Advanced science* 8, 12 (2021), 2100707.
[29] 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*. 475–485.
[30] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2777–2794.
[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX security symposium (USENIX security 19)*. 1949–1966.
[32] J Laird Marshall and Edward H Haertel. 1975. A Single-Administration Reliability Index for Criterion-Referenced Tests: The Mean Split-Half Coefficient of Agreement. (1975).
[33] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.
[34] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.
[35] Jiradet Ounjai, Valentin Wüstholz, and Maria Christakis. 2023. Green fuzzer benchmarking. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1396–1406.
[36] Pratyush Patel, Theo Gregersen, and Thomas Anderson. 2024. An agile pathway towards carbon-aware clouds. *ACM SIGENERGY Energy Informatics Review* 4, 3 (2024), 10–17.
[37] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.
[38] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

[39] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1974–1993. doi:10.1109/SP54263.2024.00137

[40] Ian Schneider and Taylor Mattia. 2024. Carbon accounting in the Cloud: a methodology for allocating emissions across data center users. *arXiv preprint arXiv:2406.09645* (2024).

[41] Patrick Schober, Christa Boer, and Lothar Schwarte. 2018. Correlation Coefficients: Appropriate Use and Interpretation. *Anesthesia & Analgesia* 126 (02 2018), 1. doi:10.1213/ANE.0000000000002864

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

[43] Kostya Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX, Boston, MA, USA, 309–318. https://api.semanticscholar.org/CorpusID:11024896

[44] Charles Spearman. 1961. The proof and measurement of association between two things. (1961).

[45] Robert Swiecki. 2021. Honggfuzz. https://github.com/google/honggfuzz Online; accessed 10-October-2024.

[46] Howard EA Tinsley and Steven D Brown. 2000. Handbook of applied multivariate statistics and mathematical modeling. Academic press.

[47] Howard EA Tinsley and David J Weiss. 2000. Interrater reliability and agreement. In *Handbook of applied multivariate statistics and mathematical modeling*. Elsevier, 95–124.

[48] Matthijs J Warrens. 2015. On Cronbach's alpha as the mean of all split-half reliabilities. In *Quantitative Psychology Research: The 78th Annual Meeting of the Psychometric Society*. Springer, 293–300.

[49] Yi Wei, Bertrand Meyer, and Manuel Oriol. 2008. *Is branch coverage a good measure of testing effectiveness?* Springer, 194–212.

[50] World Resources Institute. 2004. The Greenhouse Gas Protocol. *World Resources Institute and World Business Council for Sustainable Development: Washington, DC, USA* (2004).

[51] Michał Zalewski. 2021. american fuzzy lop. https://lcamtuf.coredump.cx/afl/. [Online; accessed 10.Oct.2024].

[52] Andreas Zeller and Sascha Just. 2019. When Results Are All That Matters: Consequences. https://fuzzing-survey.org/blog/when-results-are-all-that-matters-consequences. Accessed: 2026-01-11.

[53] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3699–3715. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

[54] Qingqing Zhu and Patricia A. Lowe. 2018. *The SAGE Encyclopedia of Educational Research, Measurement, and Evaluation.* SAGE, 2455 Teller Road, Thousand Oaks, CA 91320, USA, Chapter Split-Half Reliability, pages 1573–1574; pages 1573–1574.

Received 2025-09-12; accepted 2025-12-22