Green Fuzzer Benchmarking

Jiradet Ounjai
jounjai@mpi-sws.org
MPI-SWS
Germany

Valentin Wüstholz
valentin.wustholz@consensys.net
ConsenSys
Austria

Maria Christakis
maria.christakis@tuwien.ac.at
TU Wien
Austria

ABSTRACT

Over the last decade, fuzzing has been increasingly gaining traction due to its effectiveness in finding bugs. Nevertheless, fuzzer evaluations have been challenging during this time, mainly due to lack of standardized benchmarking. Aiming to alleviate this issue, in 2020, Google released FuzzBench, an open-source benchmarking platform, that is widely used for accurate fuzzer benchmarking.

However, a typical FuzzBench experiment takes CPU years to run. If we additionally consider that fuzzers under active development evaluate any changes empirically, benchmarking becomes prohibitive both in terms of computational resources and time. In this paper, we propose GreenBench, a greener benchmarking platform that, compared to FuzzBench, significantly speeds up fuzzer evaluations while maintaining very high accuracy.

In contrast to FuzzBench, GreenBench drastically increases the number of benchmarks while drastically decreasing the duration of fuzzing campaigns. As a result, the fuzzer rankings generated by GreenBench are almost as accurate as those by FuzzBench (with very high correlation), but GreenBench is from 18 to 61 times faster. We discuss the implications of these findings for the fuzzing community.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzzing, testing, benchmarking

ACM Reference Format:
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 (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3597926.3598144

1 INTRODUCTION

Greybox fuzzing [4, 6] has shown to be effective in finding bugs, thereby improving software quality. In the last decade, it has seen wide industrial adoption and significant research advancements. For instance, Google's OSS-Fuzz service [5] has found 28,000 bugs across 850 open-source projects using various fuzzers [3, 4, 6, 16], and, in 2022 alone, the four top software-engineering conferences (ASE, FSE, ICSE, ISSTA) published 36 papers containing "fuzz" in their title.

Over the years however, evaluation of fuzzing techniques has been challenging, mainly due to lack of standard benchmarking platforms, metrics, and benchmarks. In early 2020, FuzzBench [26], an open-source benchmarking service, was released by Google to alleviate such issues. A FuzzBench experiment typically compares about 11 fuzzers; each fuzzer is run on about 20 real-world benchmark programs; each run involves 20 fuzzing campaigns (i.e., trials) of 23 hours. All raw data is made available to the user together with a result report showing statistically significant comparisons among fuzzers, i.e., fuzzer rankings. FuzzBench is currently widely used for accurate fuzzer benchmarking.

However, FuzzBench experiments are extremely costly, in terms of both computational resources and time. It is certainly prohibitive to regularly run such experiments on "academic-scale" infrastructure to evaluate improvements to a fuzzer under development. And even though researchers may request FuzzBench experiments to be run on Google's infrastructure for free, these still take days to complete. Overall, a research project in this area could require CPU centuries and tens of thousands of dollars [26], regardless of whether this money is spent by Google.

Setting researcher time aside for the moment, the computational time needed for fuzzer benchmarking raises significant environmental and financial concerns. The global energy crisis has increased the cost of such experiments, and many industrial and academic resources are already under severe scrutiny by cost-saving measures. So, on the one hand, more comprehensive and statistically significant fuzzer evaluations advance the state of the art. On the other hand however, we can't afford them especially since fuzzer improvements are the result of many iterations and intermediate experiments that guide research and development efforts.

Our approach. In this paper, we propose GreenBench, a green benchmarking platform that aims to run comprehensive fuzzer evaluations in a fraction of the resources and time required by FuzzBench. The purpose of GreenBench is to compute quick and inexpensive fuzzer rankings while maintaining high accuracy with respect to FuzzBench—in fact, GreenBench's results are comparable to those of FuzzBench (with 0.82 correlation in our experiments).

We are exploring a trade-off here, between speed and accuracy in ranking fuzzers. With GreenBench, users can obtain highly accurate results without spending prohibitive amounts of time as until now. However, as expected, GreenBench's results are not perfectly accurate with respect to FuzzBench, which is why GreenBench is not meant to replace large-scale experiments—there is still potential gain from running them.

The key idea behind GreenBench is to run on a large number of benchmarks (i.e., thousands) only for a short period of time (i.e.,

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598144

minutes). This is in contrast to FuzzBench, which runs on a few programs for many hours. As a result, GreenBench generates almost the same ranking of fuzzers as FuzzBench (with a correlation of 0.82) from 18 to 61 times faster. But how do we obtain thousands of benchmarks?

GreenBench creates benchmarks by using the existing FuzzBench programs with *diverse seed inputs* (i.e., 100 seed inputs per program). As a result, each GreenBench benchmark is significantly different from others since seed inputs are known to have major impact on fuzzer effectiveness [20, 21, 29]. Intuitively, providing diverse seed inputs for a particular program is analogous to exploring the same maze from different starting positions. In contrast, FuzzBench always uses the same seed inputs for each program. We argue that this is a poor design decision independently of approach—it evaluates fuzzers only on a specific part of the input state space for each program, and therefore, it may lead to over-fitting fuzzers under active development to these particular seeds.

Hence, GreenBench evaluates the effectiveness of fuzzers on benchmarks of the form (P_i, S_{ij}), where P_i is a program and S_{ij} a diverse seed input for P_i. As with FuzzBench, the fuzzer that performs the best (with statistical significance) according to a given metric, i.e., achieved coverage or detected bugs, within the time limit wins. Given the challenges of bug-based benchmarking [12], achieved coverage is typically preferred and the default metric in FuzzBench.

When using a coverage-based metric, GreenBench implements the following optimization for larger cost savings. During fuzzing, we can of course not know when all possible coverage of a given benchmark has been achieved. Consequently, fuzzers run until the time limit even when there are no more branches to cover. However, to save even more time and energy, we can further specify our benchmarks with *target coverage*, i.e., the coverage that a fuzzer should achieve for the benchmark to be considered complete. If the fuzzer reaches the target coverage before the time limit, it stops.

We define target coverage as a set of *target (control-flow) edges* that are covered by k *target inputs* but not by seed input S_{ij}. In particular, each benchmark now becomes $(P_i, S_{ij}, \overline{E^T_{ij}})$, where $\overline{E^T_{ij}}$ is the set of target edges. Parameter k is important in controlling the difficulty of each benchmark, that is, it guarantees that the fuzzer needs to generate at least k inputs to cover all edges in $\overline{E^T_{ij}}$ independently of the size of this set.

When using target coverage to rank fuzzers, GreenBench ranks the fuzzer that covers the most target edges first. To break any ties, we additionally use the time it takes for fuzzers to cover the target edges as well as the total number of covered edges.

Contributions. Our paper makes the following contributions:

- We propose GreenBench, a novel benchmarking platform that speeds up fuzzer benchmarking by orders of magnitude, thereby saving significant time and energy.
- We implement GreenBench as an open-source extension of FuzzBench, a widely used platform for accurate fuzzer benchmarking.
- We evaluate GreenBench against FuzzBench in terms of speed and accuracy; our results show that GreenBench can generate a fuzzer ranking with very high correlation with the ranking generated by FuzzBench but from 18 to 61 times faster.
- We discuss the implications of our findings for the fuzzing community.

Outline. The rest of this paper is organized as follows. Section 2 explains and motivates our fuzzer-benchmarking approach. In Section 3, we describe our implementation of GreenBench, and in Section 4, we present our experimental evaluation comparing GreenBench with FuzzBench in terms of speed and accuracy. Section 5 elaborates on the importance of greener fuzzer benchmarking. We discuss related work in Section 6 and conclude in Section 7.

2 APPROACH

Our GreenBench approach incorporates three key changes in the FuzzBench platform:

(1) Randomizing initial seed inputs of benchmark programs, thereby creating benchmarks of the form (P_i, S_{ij});
(2) Drastically reducing the duration of fuzzing campaigns (from 23 hours to 15 minutes in our default configuration);
(3) Drastically increasing the number of campaigns per benchmark program (from 20 campaigns, each running on the same P_i with the same, fixed seed input S_i, to 100 campaigns in our default configuration, each running on the same P_i but with diverse seed inputs S_{ij}, where $j = 1 \ldots 100$).

Figure 1 illustrates these changes visually. The outer black oval represents the input state space of a given benchmark program P_i. For simplicity, since the input state space is typically unbounded, a point in this oval represents many inputs that exercise the same program path. FuzzBench runs many fuzzing campaigns from the same seed S_i. The inputs discovered during those campaigns are depicted by the red-shaded area. As fuzzers are non-deterministic, different campaigns naturally discover different sets of inputs. In the figure, the inputs within the red line represent the intersection of these sets, i.e., the inputs discovered by all campaigns.

The first two changes in GreenBench are visualized in green in the figure. First, there are many diverse initial seeds S_{ij}, and second, each campaign is shorter, thus covering a smaller part of the input state space. The third change is not explicitly visualized, but intuitively, many campaigns starting from diverse seeds aim to evenly cover the input state space with green areas. Note that starting a campaign with S_{i5} allows evaluating a fuzzer's effectiveness on a region of the input space that would hardly be reached when only starting campaigns with S_i.

To further reduce costs and optimize experiment running time, we additionally propose a fourth change. Instead of comparing fuzzer effectiveness with respect to the total achieved edge coverage, we suggest to base the comparison on the achieved coverage of certain randomly selected target edges. By bounding the number of edges to be covered, a fuzzing campaign can be terminated as soon as all target edges are covered instead of allowing it to reach the time limit. As shown in our experiments, this optimization indeed results in larger time savings without sacrificing accuracy of the benchmarking platform.

In the following, we describe all four changes in more detail and discuss the motivation behind these design choices.

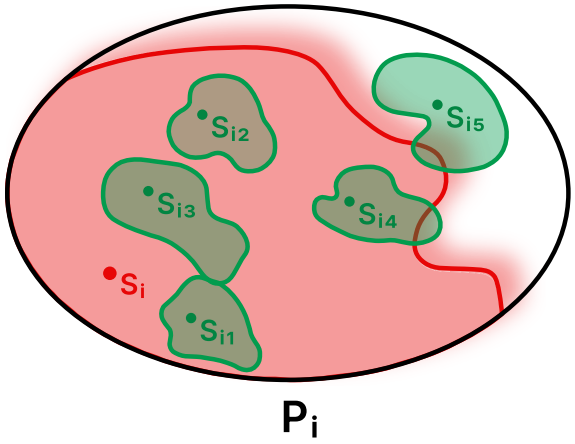

Figure 1: Visual comparison of GreenBench (in green) and FuzzBench (in red) with respect to the input state space of a given benchmark program P_i (in black). FuzzBench uses the same set of initial seeds S_i for all campaigns. In contrast, GreenBench randomly uses diverse initial corpora S_{ij} to run significantly more short campaigns.

2.1 Randomizing Initial Seed Inputs

Seed inputs can have significant impact on the effectiveness of a fuzzing campaign [20, 21, 29], and as a result, the fuzzing community has been debating what seeds to use for benchmarking [21]. There are two extreme options, namely, using an empty seed corpus or a large, almost saturated corpus.

In practice, an empty corpus is mainly relevant for fuzzing a new program with no existing seed inputs. During the first few hours of a campaign, most fuzzers will discover many new edges. Using an empty corpus is, therefore, not suitable for comparing fuzzers with a relatively short time limit as they will all quickly increase the achieved coverage and the variance of their effectiveness will be high.

In contrast, a large corpus can be used for programs that have already been extensively fuzzed. In such cases, increasing coverage is difficult, and any newly discovered edges provide a good indication of a fuzzer's effectiveness. However, it may also bias comparisons in favor of fuzzers that specialize in finding new coverage using more expensive techniques. In addition, there may not be any noticeable coverage increase in a long period of time, thus rendering a very large corpus less suitable for benchmarking.

FuzzBench balances these two extremes by using a range of corpus sizes across different benchmark programs. However, the corpus is fixed for each program, and consequently, there is a risk of over-fitting fuzzers as we will see in our experimental evaluation. More specifically, a fixed corpus may bias comparisons in favor of fuzzers that are under active development and use FuzzBench to validate algorithmic choices or tune hyper-parameters.

To reduce such bias, we randomize the initial seed corpus for different campaigns run on the same benchmark program. By default, GreenBench uses a single random seed input, but it may also be configured to use a corpus of any size. We pick seed inputs from an existing large corpus uniformly at random. The large corpus can be obtained by running a long fuzzing campaign once per benchmark program and using the generated pool as the corpus, or by directly using Google's OSS-Fuzz corpus [5]. Since seed selection is performed randomly, each fuzzing campaign will (most likely) sample a different part of the input state space—by construction, each seed is guaranteed to cover a different program path since this is the criterion that AFL-like fuzzers use for adding seeds to their corpus. This is most noticeable at the beginning of fuzzing (i.e., first minutes or hours) when there is typically little coverage overlap between two campaigns started from diverse inputs.

As shown in Figure 1, using a diverse initial corpus for each campaign explores the input state space more broadly right from the start. Not only does this change reduce the risk of over-fitting, but it also allows for drastically reducing the campaign duration—no time needs to be spent on discovering the same inputs over and over again across campaigns.

2.2 Drastically Reducing Campaign Duration

Over the last years, fuzzing campaigns have been established to be relatively long, i.e., lasting one or more days. FuzzBench is no exception—its campaigns are 23 hours long (they are not 24-hour campaigns to reduce costs by running on less expensive cloud instances). This practice is mainly motivated by the fact that coverage variance tends to decrease with time, and of course, less variance tends to provide more reliable comparisons among fuzzers. However, the concrete choice of the time limit is not well motivated since variance depends on the benchmark program, i.e., variance may decrease more quickly for some benchmarks than others. So, one way to reduce costs would be to better calibrate the time limit for each benchmark program.

Our approach is even bolder by using a very short time limit (15 minutes by default) across all benchmark programs. Viewed in isolation, this change seems like a poor design decision—our experimental results also substantiate it as such. However, it should be considered in combination with change 1 (randomizing initial seed inputs) and change 3 (drastically increasing campaign number). As shown in Figure 1, by using a large number of short campaigns, each evaluating fuzzers on a different part of the input state space, we do not waste time re-discovering the same inputs. Instead, fuzzers are evaluated on diverse seed corpora and may reach parts of the input space that they would not with FuzzBench.

2.3 Drastically Increasing Campaign Number

In fuzzer benchmarking, it is customary to run many campaigns per benchmark program since fuzzers are non-deterministic tools. In other words, the final achieved coverage by two campaigns of the same fuzzer (with the same seed inputs and benchmark program) may vary. FuzzBench by default runs 20 campaigns per benchmark program, thus allowing to compute statistical measures, such as variance, and to compare statistical significance of any differences in the effectiveness among fuzzers.

In practice however, the final coverage achieved with FuzzBench has very low variance for most benchmark programs and fuzzers. This is due to the long campaign duration as well as due to the use of the same seed corpus for all campaigns on a given benchmark

program. Consequently, running 20 such campaigns is a waste of resources, thereby creating an opportunity for cost savings.

Even though GreenBench only runs a single campaign for each *benchmark*, i.e., a benchmark program and its randomized seed corpus, it runs significantly more campaigns for the same *benchmark program*. The key idea is to rely on more campaigns to ensure that the short campaign duration (change 2) still evaluates the fuzzer for a large space of interesting inputs. Intuitively, many randomly distributed, small, green areas in Figure 1 cover the black oval more evenly than a single, large, red area.

2.4 Bounding Target Edge Coverage

Fuzzer benchmarking typically uses two measures of effectiveness, namely achieved code coverage and detected bugs. Due to a number of challenges with using the number of detected bugs [12], achieved coverage is a more established metric. (We use code coverage here even though GreenBench could easily be adapted to use the number of detected bugs instead.) However, it is not tractable to determine the maximum possible coverage for real-world benchmarks. Otherwise, fuzzing campaigns could be terminated once this maximum was reached, allowing to further reduce costs.

In GreenBench, we design an approximate solution that enables terminating early. On a high level, we randomly select a subset of all feasible edges that are not already covered by the initial seed corpus of a benchmark program, i.e., given a benchmark (P_i, S_{ij}), we select edges in P_i that are not covered by S_{ij}. We refer to these edges as *target edges* and terminate a campaign as soon as it covers all target edges. This change of randomly selecting a subset of all edges resembles bug-based benchmarking since bugs also occur sparsely in a program.

Under the hood, we first need to approximate the set of feasible edges $\overline{E}$ that are not already covered by the initial seed corpus. Such a set can be determined using the existing large corpus, which we also needed for randomizing initial seed inputs (change 1). $\overline{E}$ is then the set of edges covered by the large corpus excluding those covered by the initial corpus. Even though the target edges $\overline{E^T}$ are a subset of $\overline{E}$, we cannot simply uniformly sample from $\overline{E}$ to form $\overline{E^T}$. This is because we may include edges that are very difficult to cover within the short time limit of fuzzing campaigns, consequently defeating the purpose of saving costs and producing a meaningful comparison among fuzzers.

Instead, GreenBench implements the following alternative. We randomly select k inputs from the large corpus that are not already in the initial corpus. $\overline{E^T}$ is then composed of those edges that are not covered by the initial corpus but are covered by the k random inputs. Algorithm 1 shows our approach for generating a benchmark of the form $(P_i, S_{ij}, \overline{E^T_{ij}})$ for a benchmark program P_i when given an existing large *corpus* and parameter k.

Lines 2–3 randomly select an initial seed input from the large *corpus* and determine the coverage it achieves in the benchmark program. Next, lines 4–7 randomly select k target inputs and determine their coverage. The final target coverage is computed by removing any edges that are already covered by the initial seed input (line 8). If the final target coverage is non-empty (line 9),

Algorithm 1: GreenBench's benchmark-generation algorithm for a given benchmark program P, a large set of potential initial seed inputs *corpus*, and a number k of target inputs. On a high level, we first randomly select an initial seed S and k target inputs from the *corpus*. The benchmark then consists in program P, the initial seed S, and the set of edges that are covered by the target inputs but are not covered by S.

```
1  Function generateRandomBenchmark(P, corpus, k):
       /* randomly pick initial seed input S from corpus */
2      S := randomPick(corpus);
       /* compute the edges E^S that S covers in P */
3      E^S := coveredEdges(P, S);
       /* initialize the target edges E^T to an empty set */
4      E^T := ∅;
5      repeat k times
           /* randomly pick a target input T from corpus */
6          T := randomPick(corpus);
           /* add the edges that T covers in P to E^T */
7          E^T := E^T ∪ coveredEdges(P, T);
       /* remove from E^T any edges already covered by S */
8      E^T := E^T \ E^S;
       /* if E^T is non-empty */
9      if E^T ≠ ∅ then
           /* return the new benchmark */
10         return (P, S, E^T);
       /* otherwise try again */
11     return generateRandomBenchmark(P, corpus, k)
```

a new benchmark is returned (line 10), otherwise the algorithm attempts to generate another random benchmark (line 11).

This approach of selecting target edges bounds the difficulty of the generated benchmarks as k inputs suffice for covering all target edges. It also allows for a smooth increase in difficulty for a given benchmark. In particular, GreenBench gives partial credit to fuzzers for covering only some of the target edges, e.g., the shallower (and therefore easier) ones.

When the same number of target edges is covered by multiple fuzzers, GreenBench breaks the tie by, first, using the time to find the target edges, and finally, the total number of covered edges for each fuzzer.

3 IMPLEMENTATION

Our implementation reuses and extends the existing FuzzBench infrastructure (e.g., benchmark programs and fuzzer runners) as much as possible. This section provides a short overview of the most important implementation changes, some of which are incorporated in the mainline FuzzBench project.

First, we made it possible to compare fuzzers based on edge coverage—previously, FuzzBench used region coverage. Edge coverage is a more common and well understood metric, and is now the default in FuzzBench.

Second, we added a feature in FuzzBench to provide custom initial seed inputs (instead of the fixed seed corpus) for different benchmark programs and campaigns. In GreenBench, we use this feature to start campaigns with randomized seed inputs.

Third, we extended the coverage-measurement module to keep track of target-edge coverage in addition to measuring the overall edge coverage.

Moreover, in our implementation, we adjusted several FuzzBench settings. First, we use a coverage-measurement interval of 1 minute instead of 15 minutes. Second, we reduce the time limit for campaigns from 23 hours to 15 minutes. Third, we increase the number of campaigns per benchmark program from 20 to 100. These are our default settings, but we also consider several variants in our experimental evaluation.

4 EXPERIMENTAL EVALUATION

In this section, we address the following research questions:

- **RQ1:** How long does it take to generate random benchmarks of the form $(P_i, S_{ij}, \overline{E_{ij}^T})$ from existing benchmark programs P_i and a large seed corpus?
- **RQ2:** How much time does GreenBench save with respect to FuzzBench?
- **RQ3:** How accurate is GreenBench versus FuzzBench?
- **RQ4:** Can GreenBench run fewer campaigns without sacrificing accuracy?
- **RQ5:** Can GreenBench run shorter campaigns without sacrificing accuracy?
- **RQ6:** Are GreenBench results stable?

4.1 Setup

Large corpora. Recall that large corpora are needed in changes 1 and 4 of GreenBench. We used AFL++ [16] (commit 45668bb) to generate a large corpus for each benchmark program. For generating a corpus, we used the default settings from the AFL++ FuzzBench setup. We selected AFL++ for this purpose since it was the winning fuzzer in the FuzzBench paper [26], but we could have also used another fuzzer.

Fuzzers. Due to the large computational cost of our experimental evaluation, we used a subset of six diverse fuzzers (namely, AFL [6], AFL++ [16], Eclipser [14], Entropic [11], Honggfuzz [3], and libfuzzer [4]) instead of all eleven fuzzers benchmarked in the FuzzBench paper.

Configurations. We used commit e816b71 of FuzzBench for our comparisons. Our configurations for GreenBench are described in detail in the rest of this section.

Machine. We performed all experiments on a 32-core Intel Xeon E5-2667 v2 CPU (3.30GHz) machine with 256GB of memory, running Debian GNU/Linux 11.

4.2 Results

We now discuss our findings for each research question.

RQ1: How long does it take to generate random benchmarks of the form $(P_i, S_{ij}, \overline{E_{ij}^T})$ from existing benchmark programs P_i and a large seed corpus? Figure 2 shows how long it takes to generate 100 benchmarks of the form $(P_i, S_{ij}, \overline{E_{ij}^T})$ for each of the benchmark programs, P_i, in FuzzBench and a corresponding large seed corpus. As shown in the figure, for most benchmark programs, the time is less than 5 minutes, whereas for two programs, more time is spent than for all others together. For these two outliers, the average running time per input is much higher than for other programs. The total time for all programs is 120.28 minutes, and the majority of this time is spent on executing inputs to compute their achieved edge coverage.

Note that the time for benchmark generation does not have to be spent when reusing benchmarks across experiments, which is the most common use case. In addition, note that we do not consider the time to obtain the large corpus needed for changes 1 and 4 in this research question. For each benchmark program, we built a large corpus by running a single fuzzing campaign (with AFL++), however such a corpus may also be obtained differently.

> Given a benchmark program P_i and a corresponding large corpus, it typically only takes a few minutes to generate 100 benchmarks of the form $(P_i, S_{ij}, \overline{E_{ij}^T})$.

RQ2: How much time does GreenBench save with respect to FuzzBench? A regular FuzzBench experiment with 6 fuzzers takes 55,200 CPU hours (6 fuzzers x 20 benchmark programs x 20 campaigns x 23 CPU hours), which is approximately 6.4 CPU years. In contrast, the corresponding GreenBench experiment without change 4 takes only 3,000 CPU hours (6 fuzzers x 20 benchmark programs x 100 benchmarks x 0.25 CPU hours), which is approximately 4.2 CPU months. This constitutes a *speedup of 18.4x*.

We have also investigated the additional savings of GreenBench by enabling change 4. In general, the savings depend on the number of target edges. Fewer target edges should result in more savings, possibly at the expense of accuracy (see RQ3 below). The number of target edges can be controlled by changing the number of target inputs k, i.e., larger values of k should result in more target edges. We have compared different values of k to explore how the savings decrease as k increases.

For $k = 2$, the running time of a GreenBench experiment is further reduced by 31.92%. However, this comes at the cost of significantly reduced accuracy (see RQ3). For $k = 3$, the reduction is 12.70%, and for $k = 5$, the time is reduced by 8.89%. Both of these settings have good accuracy. By setting k to a much higher value ($k = 50$), the reduction is only 3.35% without notably increasing accuracy. Our default configuration with $k = 5$ and change 4 enabled provides a *speedup of 20.2x* over FuzzBench.

As we discuss in the following research questions, there are a number of important hyper-parameters that can further affect speedup. For instance, GreenBench could even achieve a speedup

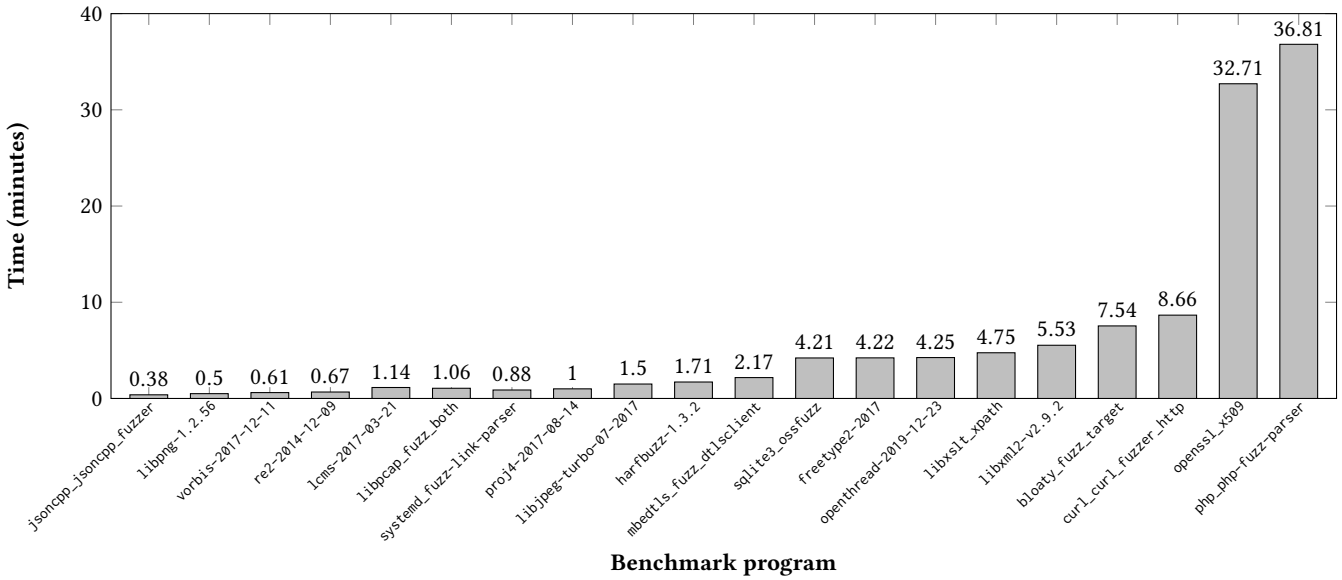

Figure 2: Benchmark-generation time for different benchmark programs. The bar chart shows how long it takes to generate 100 benchmarks for the given programs. The majority of time is spent on executing inputs to obtain edge-coverage information.

of 61.3x over FuzzBench by reducing the number of benchmarks from 100 to 30 without significantly sacrificing accuracy (see RQ4).

> The default GreenBench configuration runs in 3.8 CPU months, in contrast to FuzzBench, which takes 6.4 CPU years. There are GreenBench configurations that can even bring its running time down to 37.5 CPU days without significantly sacrificing accuracy.

RQ3: How accurate is GreenBench in comparison with FuzzBench? In this research question, we investigate the accuracy of GreenBench (and each of its design choices) by comparing the correlation of its fuzzer ranking with the FuzzBench ranking—we use the standard ranking function from FuzzBench [26]. We consider the following nine configurations:

- **FB:** The vanilla FuzzBench configuration that runs 20 campaigns per benchmark program, each of 24 hours and with the same initial seed corpus[1];
- **R:** A variant of FB that applies change 1 of randomizing initial seed inputs, i.e., it runs 20 campaigns per benchmark program, each of 24 hours but with a randomized initial seed input;
- **RD:** A variant of R that additionally applies change 2 of drastically reducing the campaign duration, i.e., it runs 20 campaigns per benchmark program, each of 15 minutes and with a randomized initial seed input;
- **RDN:** A variant of RD that additionally applies change 3 of drastically increasing the campaign number, i.e., it runs 100 campaigns per benchmark program, each of 15 minutes and with a randomized initial seed input;
- **RDN-2:** A variant of RDN that additionally applies change 4 of bounding the target edge coverage with $k = 2$;
- **RDN-3:** A variant of RDN that additionally applies change 4 with $k = 3$;
- **RDN-5:** A variant of RDN that additionally applies change 4 with $k = 5$;
- **RDN-50:** A variant of RDN that additionally applies change 4 with $k = 50$;

Table 1 shows the fuzzer rankings that are produced by these different configurations and Table 2 the correlation between the fuzzer rankings of all configurations.

When comparing FB and R, the correlation (Table 2) drops to 0.89, confirming the substantial effect of initial seeds on benchmarking results. Certain fuzzers, such as AFL++ and Eclipser, seem to significantly benefit from the fixed seed corpus (Table 1). In fact, AFL++ seems to have been extensively tuned using FuzzBench experiments[2], which could explain why there is over-fitting to these specific seeds. To reduce such potential bias, we will use R as our main baseline. A very recent study on explainable fuzzer evaluation [28] independently makes a similar observation and tries to explain a fuzzer ranking through properties, such as size or coverage, of the initial corpus or of the benchmark programs.

When comparing R and RD, the correlation drops significantly, to 0.60. This is, of course, not surprising and confirms that 20 short campaigns per benchmark program are not able to reliably cover their input state space. By increasing the number of campaigns to

[1]Note that, for technical reasons, FuzzBench runs campaigns of 23 hours instead of 24. For our evaluation, we follow the commonly used recommendation of running for 24 hours [21].

[2]See, for instance, the > 150 public benchmark results with `aflpp` and `aflplusplus` in their directory name under https://www.fuzzbench.com/reports/experimental/index.html (accessed on May 5, 2023). Many of these experiments compare several variants of AFL++; consider the computational, environmental, and financial concerns raised by these experiments.

Table 1: Fuzzer ranking for different benchmarking configurations.

Fuzzer \ Config	FB	R	RD	RDN	RDN-2	RDN-3	RDN-5	RDN-50
AFL	4.20	4.30	4.25	3.58	3.87	3.51	3.68	3.60
AFL++	2.25	2.45	2.65	2.35	2.90	2.90	2.89	2.91
Eclipser	3.20	3.80	4.25	4.20	3.96	3.72	3.75	3.75
Entropic	3.10	2.33	3.42	3.28	3.36	3.52	3.41	3.47
Honggfuzz	3.43	3.20	2.88	3.02	2.98	3.25	3.14	3.22
libFuzzer	4.83	4.93	3.55	4.58	3.66	3.88	3.96	3.96

Table 2: Correlation between the fuzzer rankings that are produced by different benchmarking configurations. The Pearson correlation coefficient ranges from -1 (no correlation) to 1 (perfect correlation). Values in the $[0.8, 1]$ interval are commonly considered to indicate very strong correlation, and values in the $[0.6, 0.8)$ interval are commonly considered to indicate strong correlation.

Config	FB	R	RD	RDN	RDN-2	RDN-3	RDN-5	RDN-50
FB	1.00							
R	0.89	1.00						
RD	0.50	0.60	1.00					
RDN	0.78	0.82	0.71	1.00				
RDN-2	0.58	0.69	0.98	0.82	1.00			
RDN-3	0.77	0.72	0.73	0.97	0.82	1.00		
RDN-5	0.82	0.83	0.81	0.97	0.90	0.97	1.00	
RDN-50	0.81	0.80	0.76	0.98	0.85	0.99	0.99	1.00

100 in RDN, the correlation with R increases substantially, to 0.82. Therefore, more campaigns compensate for their short duration.

Let us now evaluate the effect of change 4 by considering different values for parameter k, namely, 2, 3, 5, and 50. We observe that the correlation with R drops for $k = 2$, but it increases again as we increase k. For $k = 5$, the correlation even surpasses the RDN configuration, thereby improving both accuracy and time savings. Notice that $k = 50$ does not increase accuracy with respect to R while also saving less time in comparison with smaller k values (see RQ2). We, therefore, consider RDN-5 the default GreenBench configuration.

The fixed initial seeds of FuzzBench may lead to over-fitting.

The fuzzer ranking generated by the default GreenBench configuration has an 0.83 correlation with the ranking generated by FuzzBench when randomizing the initial seeds, but it is computed 20 times faster.

RQ4: Can GreenBench run fewer campaigns without sacrificing accuracy? In our default configuration, RDN-5, we use 100 campaigns per benchmark program. However, this number could potentially be reduced further without sacrificing accuracy. In this research question, we investigate how the choice of this parameter affects accuracy.

Figure 3 plots the correlation in fuzzer ranking for different campaign numbers ($c = 1, \ldots, 100$) with respect to baseline R. Recall that, for each benchmark program P_i, GreenBench has generated 100 benchmarks of the form $(P_i, S_{ij}, \overline{E_{ij}^T})$. In this experiment, for every value of c, we shuffle these benchmarks 100 times, and each time, we select the first c benchmarks for fuzzing. In the figure, we compute the median correlation with R (dark line) and determine 95%-confidence intervals (shaded area).

Even with lower values for c, such as 30, we already obtain a similar median correlation as our default configuration. In fact, for $c = 28$, the correlation is already 0.83—the same as for RDN-5. This demonstrates that GreenBench could, in principle, save even more time: when changing RDN-5 to set $c = 30$, GreenBench is over 61x faster than FuzzBench. Overall, the number of campaigns provides a knob for controlling the accuracy-vs-speed trade-off.

GreenBench could run as few as 28 campaigns per benchmark program without sacrificing its accuracy while being 61 times faster than FuzzBench.

RQ5: Can GreenBench run shorter campaigns without sacrificing accuracy? In our default configuration, RDN-5, we use a time limit of 15 minutes (or 900 seconds) for each campaign. In this research question, we investigate how this time limit affects

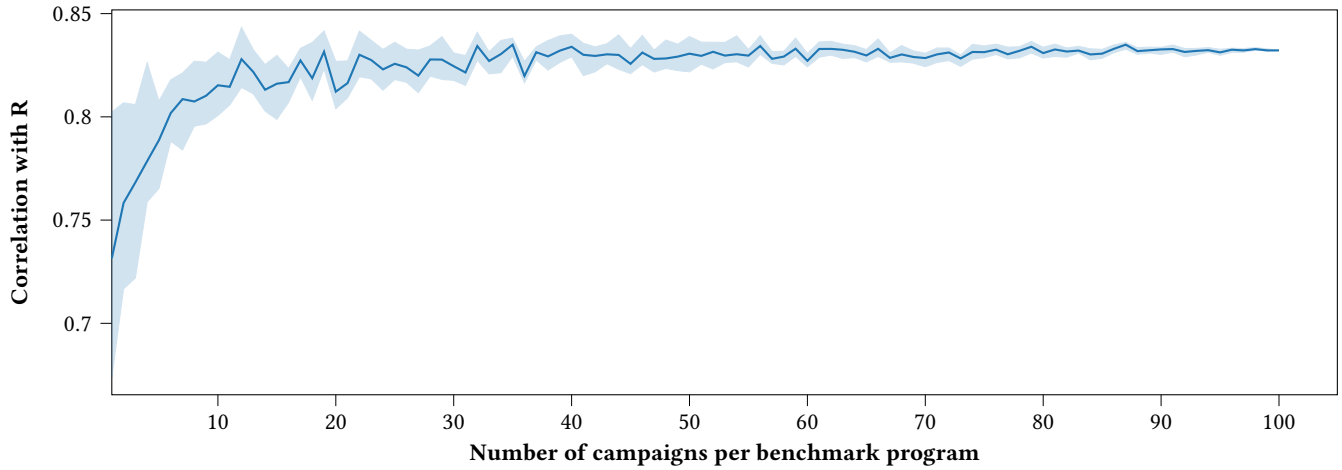

Figure 3: Correlation with baseline configuration R for different values of c (number of campaigns per benchmark program). After about 30 campaigns, the correlation only improves minimally.

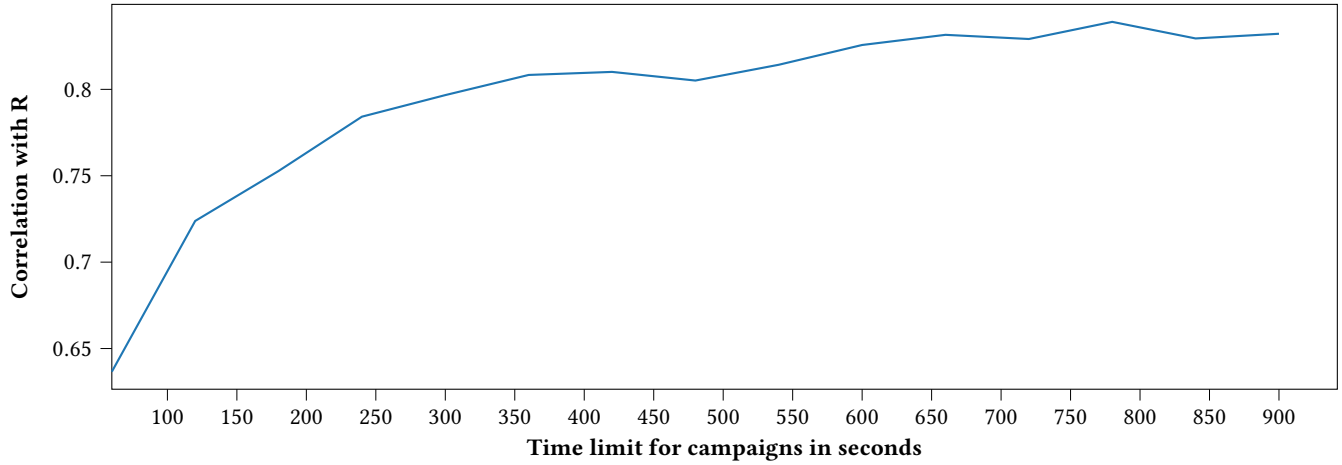

Figure 4: Correlation with baseline configuration R for different values of l (time limit for campaigns). After about 10 minutes, the correlation only improves minimally.

accuracy, and in particular, by how much it could be shortened without sacrificing accuracy.

Figure 4 plots the correlation in fuzzer ranking for different time limits ($l = 60, \ldots, 900$ seconds) with respect to baseline R. As shown in the figure, the correlation converges quickly, and after about 10 minutes (or 600 seconds), it barely changes. In fact, at 600 seconds, the correlation is already 0.83—the same as for RDN-5. Again, GreenBench could save even more time: when changing RDN-5 to set $l = 600$, GreenBench is over 27x faster than FuzzBench. We also experimented with time limits of up to 30 minutes, and the correlation did not increase further. In general, the campaign duration also controls the accuracy-vs-speed trade-off and could be dynamically adjusted for different benchmark programs.

GreenBench could reduce the campaign duration down to 10 minutes without sacrificing its accuracy while being 27 times faster than FuzzBench.

RQ6: Are GreenBench results stable? Since fuzzers are non-deterministic, a benchmarking experiment may generate slightly different results from another. To investigate the stability of GreenBench results across different benchmarking experiments, we performed three independent runs of our default configuration, RDN-5, on three different machines with the same hardware configuration.

Table 3 shows the fuzzer rankings that are produced by these three runs and Table 4 their correlation. We observe that all runs have very high correlation. In fact, runs 1 and 3 generate the same

Table 3: Fuzzer rankings for three independent benchmarking runs with our default configuration (RDN-5). We also show the ranking for baseline configuration R for comparison.

Fuzzer \ Config	R	RDN-5		
		Run 1	Run 2	Run 3
AFL	4.30	3.68	3.45	3.52
AFL++	2.45	2.89	2.84	2.85
Eclipser	3.80	3.75	3.66	3.69
Entropic	2.33	3.41	3.50	3.48
Honggfuzz	3.20	3.14	3.34	3.31
libFuzzer	4.93	3.96	4.01	3.98

Table 4: Correlation between the fuzzer rankings for three independent benchmarking runs with our default configuration (RDN-5). We also show the correlation with baseline configuration R for comparison.

Config		R	RDN-5		
			Run 1	Run 2	Run 3
R		1.00			
RDN-5	**Run 1**	0.83	1.00		
	Run 2	0.74	0.93	1.00	
	Run 3	0.77	0.96	0.99	1.00

fuzzer ranking. Run 2 also agrees with the exception of ranking Entropic and AFL in the reverse order.

Multiple repetitions of each independent run can obviously help to obtain even more stable results. However, such a design choice comes at a cost and, as we argue over Figure 1, the time might be better spent on running with different seed inputs, e.g., 100 campaigns by default in GreenBench.

> The results of independent GreenBench experiments have very high correlation.

4.3 Threats to Validity

We have identified the following threats to the validity of our experiments.

Benchmark programs. The choice of benchmark programs is important when evaluating fuzzers [21] as well as benchmarking platforms. For our experiments, we used 20 benchmark programs from the FuzzBench platform. These are diverse, well established programs from various application domains and have already been used in multiple fuzzer evaluations. However, our results may not generalize to a different selection of benchmark programs.

Fuzzers. Due to the large computational cost, we used a subset of six fuzzers for our experiments (namely, AFL [6], AFL++ [16], Eclipser [14], Entropic [11], Honggfuzz [3], and libfuzzer [4]) instead of all eleven fuzzers from the FuzzBench paper. We tried to select a diverse subset, but our results may not generalize to a different selection of fuzzers. Moreover, by building on FuzzBench, GreenBench is as applicable to different fuzzers as FuzzBench.

Large corpus. Our approach uses a large, but fixed, corpus of seed inputs per benchmark program. These inputs are used both for randomizing the initial corpus for our benchmarks and for selecting target edges. Therefore, they may influence the accuracy of our approach. We used the winning fuzzer from the FuzzBench paper (AFL++) to generate these corpora. AFL++ only adds inputs to its corpus when they increase coverage, thereby guaranteeing a diverse set of inputs. However, our results may not generalize to a different seed corpus.

Choice of GreenBench parameters. The choice of k (number of target inputs), c (number of campaigns per benchmark program), and l (time limit per campaign) can influence the accuracy and performance of our approach. To mitigate this threat, we have evaluated our approach using a range of values for all these parameters (see RQ2–5). However, our results may not generalize to different choices of GreenBench parameters.

Fuzzer non-determinism. Since fuzzers are non-deterministic, one benchmarking run may produce slightly different results from another. To mitigate this threat, we ran our default-configuration experiment three times (see RQ6).

Effectiveness metric. We use code coverage as our main effectiveness metric for fuzzers. An alternative would be to use the number of detected bugs—in fact, changes 1–3 are directly applicable to bug-based evaluations when appropriately adjusting the campaign duration such that bugs are found. However, coverage is used more often in practice since bugs are rare in real-world code. A recent study [12] discusses some of the challenges with using bugs as an effectiveness metric, and in any case, found that there is very high correlation between the two metrics. Nevertheless, our results may not generalize to a different effectiveness metric.

5 DISCUSSION

Is GreenBench green? GreenBench brings us a significant step forward in mitigating environmental concerns with fuzzer benchmarking. It allows fuzzer developers to speed up evaluations by orders of magnitude, and in addition, it provides knobs to adjust the accuracy guarantees depending on the stage of fuzzer development (e.g., when evaluating a pull request, or when preparing a new release). On the other hand, GreenBench does not completely eliminate environmental concerns, and more research is needed.

Implications for the fuzzing community. While we have highlighted environmental concerns, the main issue with current fuzzer benchmarking is multi-faceted, and raises economic, social, and methodological questions, such as:

- Will fuzzer developers with limited financial resources still be able to publish their research at top venues?
- Will the success of fuzzers hinge on the ability to run huge numbers of experiments?

- How will a fuzzer be able to beat the state of the art without years of expensive hyper-parameter tuning?
- Will we end up with a fuzzer mono-culture consisting of minor tweaks to AFL++?
- How can artifact-evaluation committees reproduce fuzzer evaluations within short review periods and on a tight budget?
- How can we prevent fuzzer developers from over-fitting to specific, well established benchmarks?

To continue to thrive, the fuzzing community should try to engage with these questions.

Going forward. Looking beyond our community, we can observe similar trends in machine learning, where it seems to have become an arms race to build larger and larger models using more and more resources. This puts academic institutions and small companies at a competitive disadvantage. However, there are also practices in the machine-learning community that could inspire and benefit us. For instance, machine-learning models are typically trained and evaluated on separate datasets. Assuming that the datasets are sufficiently different, this mitigates the risk of over-fitting.

Perhaps fuzzer developers should use different sets of benchmarks during development and when running final experiments for scientific publications. A potential first step could be to use GreenBench during development and FuzzBench for preparing a paper. However, more thought is needed to develop rigorous methodologies.

In GreenBench, we have proposed to randomize the initial seed corpus for different campaigns. A logical next step would be to also randomize the benchmark programs themselves. Of course, this would require a larger selection of suitable benchmark programs. Perhaps Google's OSS-Fuzz corpus [5] (containing about 850 open-source programs) could serve as a diverse set of such programs, and GreenBench or FuzzBench could randomly select programs from this set.

Such a larger space of possible benchmarks could reduce the feasibility of systematic hyper-parameter tuning. During artifact evaluations, a different (possibly smaller) set of benchmarks could easily be generated (i.e., by providing a different random seed to GreenBench or FuzzBench) to validate that the key claims generalize beyond the benchmarks that were used by the authors.

In GreenBench, we have also proposed to run shorter campaigns. We have already discussed the reasoning behind this choice. However, there is one additional benefit we would briefly like to discuss here. Currently, benchmarking platforms focus on the final results (often after 24 or more hours of fuzzing) but tend to de-emphasize how these results are achieved. Two fuzzers that both achieve coverage X after 24 hours are considered to perform equally well, even if fuzzer A already reaches coverage X much earlier. In other words, fuzzer A may be superior, but due to the long running time, the advantage becomes less significant.

Shorter campaign duration may not be the only solution to address this issue, but it does put more emphasis on the early stages of fuzzing campaigns when most inputs tend to be discovered. While we tried to make sure that the fuzzer ranking generated by GreenBench is similar to the ranking generated by FuzzBench, the latter should not be considered as ground truth but only as part of the current state of the art, which may evolve over time.

6 RELATED WORK

There is a large body of work on fuzzing [18, 24, 27]. In this section, we only focus on fuzzer benchmarking, which is more closely related to our approach.

Guidelines for fuzzer evaluations. Over the years, several guidelines for fuzzer evaluations and benchmarking have been created; from very general guidelines for empirical evaluations [1], to more specific guidelines for randomized algorithms (including fuzzers) [7, 8], and—most recently—even specific guidelines for fuzzers [21]. The latter, for instance, highlights the importance of initial seed inputs, time limits, effectiveness metrics, and benchmark programs.

Benchmarks. The last two concerns—effectiveness metrics and benchmark programs—have also motivated the creation of several different benchmark sets for fuzzers. On the one hand, there are synthetic benchmark sets, such as LAVA [15] and Fuzzle [22]. The former is based on real-world programs where hard-to-reach bugs are added, while the latter synthesizes maze-like programs where transitions from one position to another are guarded by conditions of varying difficulty.

On the other hand, there are benchmarks—such as those in FuzzBench [2, 26], Magma [19], and UNIFUZZ [23]—that are based on real-world programs, and both real bugs or coverage can be used for comparing fuzzer effectiveness. Two recent studies [13, 17] identified significant differences between artificial/synthetic benchmarks and ones based on real-world bugs. Another recent study [12] compared the two effectiveness metrics, namely code coverage and bugs. They found that there is very high correlation between achieved coverage and found bugs, although—surprisingly—the best fuzzer in terms of coverage may not be the best fuzzer in terms of found bugs. For all of the above benchmarks (independently of the effectiveness metric), the default campaign duration is at least 23 hours. For bug-based benchmarks, such as Magma, the actual duration may be shorter if the fuzzer finds all target bugs earlier (similar to change 4). However, the worst-case resource usage is still high.

Finally, there are also efforts for porting benchmarks from the program-verification and model-checking community [9] to testing tools (including fuzzers) [10]. This may allow for comparisons beyond fuzzers; for instance, with software model checkers [25].

Properties of fuzzer rankings. A very recent study on explainable fuzzer evaluation [28] tries to explain a fuzzer ranking through properties—such as size or coverage—of the initial corpus or of the benchmark programs. Like us, they independently point out the risk of over-fitting fuzzers to specific benchmark sets, such as FuzzBench. While they aim to quantify the risk of using specific initial corpora or benchmark programs, our proposed changes aim to mitigate some of this risk.

7 CONCLUSION

We have presented GreenBench, the first benchmarking platform that aims to reduce the exceedingly large computational cost of

fuzzer benchmarking. The default configuration of GreenBench offers a speedup of 20.2x over FuzzBench, thereby enabling much faster turnaround times. GreenBench also provides a number of knobs to tune the accuracy-vs-speed trade-off, making it possible to favor speed for incremental changes (e.g., for merging a pull request) and accuracy for larger changes (e.g., before a new fuzzer release).

In future work, we plan to investigate if and how GreenBench could be used to provide efficient regression testing for fuzzers and to pin-point fuzzer weaknesses or even bugs. A first step for achieving the latter could be to suggest tailored fuzzer "challenges", that is, benchmarks for which a fuzzer's effectiveness is significantly below average for short campaigns.

We also hope that the community will start using GreenBench. This would allow us to gather additional empirical and anecdotal evidence about usage scenarios (such as regression testing) where GreenBench can reliably be used as a substitute for FuzzBench or other benchmarking tools.

8 DATA AVAILABILITY

GreenBench is available on GitHub: https://github.com/Rigorous-Software-Engineering/greenbench
Our large seed corpora are available on Zenodo: https://doi.org/10.5281/zenodo.7645179

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful and constructive feedback. This work was supported by Maria Christakis' ERC Starting grant 101076510.

REFERENCES

[1] [n. d.]. Empirical Evaluation Guidelines (ACM SIGPLAN). http://www.sigplan.org/Resources/EmpiricalEvaluation.
[2] [n. d.]. FuzzBench: Fuzzer Benchmarking as a Service. https://google.github.io/fuzzbench.
[3] [n. d.]. Honggfuzz: A Security-Oriented, Feedback-Driven, Evolutionary, Easy-to-Use Fuzzer. https://github.com/google/honggfuzz.
[4] [n. d.]. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.
[5] [n. d.]. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://google.github.io/oss-fuzz.
[6] [n. d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[7] Andrea Arcuri and Lionel C. Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *ICSE*. ACM, 1–10.
[8] Andrea Arcuri and Lionel C. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Softw. Test. Verification Reliab.* 24 (2014), 219–250. Issue 3.
[9] Dirk Beyer. 2012. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org.
[10] Dirk Beyer. 2019. Competition on Software Testing (Test-COMP). https://test-comp.sosy-lab.org.
[11] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *ESEC/FSE*. ACM, 678–689.
[12] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *ICSE*. ACM, 1621–1633.
[13] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *AsiaCCS*. ACM, 716–730.
[14] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *ICSE*. IEEE Computer Society/ACM, 736–747.
[15] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *S&P*. IEEE Computer Society, 110–121.
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *WOOT*. USENIX.
[17] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. 2020. An Empirical Study on Benchmarks of Artificial Software Vulnerabilities. *CoRR* abs/2003.09561 (2020).
[18] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *CACM* 63 (2020), 70–76. Issue 2.
[19] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Meas. Anal. Comput. Syst.* 4 (2020), 49:1–49:29. Issue 3.
[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *ISSTA*. ACM, 230–243.
[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS*. ACM, 2123–2138.
[22] Haeun Lee, Soomin Kim, and Sang Kil Cha. 2022. Fuzzle: Making a Puzzle for Fuzzers. In *ASE*. ACM, 45:1–45:12.
[23] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *Security*. USENIX, 2777–2794.
[24] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *TSE* 47 (2021), 2312–2331. Issue 11.
[25] Kenneth L. McMillan. 2018. Interpolation and Model Checking. In *Handbook of Model Checking*. Springer, 421–446.
[26] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ESEC/FSE*. ACM, 1393–1403.
[27] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *CACM* 33 (1990), 32–44. Issue 12.
[28] Dylan Wolff, Marcel Böhme, and Abhik Roychoudhury. 2022. Explainable Fuzzer Evaluation. *CoRR* abs/2212.09519 (2022).
[29] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing Without Valid Seed Inputs. In *ICSE*. IEEE Computer Society/ACM, 712–723.

Received 2023-02-16; accepted 2023-05-03