The Unbearable Randomness of Fuzzing

Dongjia Zhang Romain Malmain Andrea Oliveri Davide Balzarotti Aurelien Francillon

***Abstract*—Fuzzing has become a cornerstone of automated software testing and vulnerability discovery. However, its inherently stochastic nature poses serious challenges for evaluation and reproducibility. Moreover, the underlying sources of this randomness, often unforeseen and uncontrollable, remain largely unexplored by the scientific community.**

In this paper, we present the first comprehensive study of the different sources of randomness in fuzzing, analyzing its root causes, impact on key performance metrics, and implications for experimental design, revealing several key findings that can guide future fuzzing practitioners. We systematically analyze the sources of variability, including target behavior, environmental conditions, and fuzzer internals, and quantify their effects through large-scale experiments across diverse configurations. Using fuzzer throughput and code coverage as primary indicators, we reveal that variability persists even under tightly controlled conditions, and that conventional practices, such as fixing the PRNG seed, are insufficient to ensure consistency. Through statistical power analysis and meta-evaluation of real-world benchmarking data, we demonstrate that commonly adopted experimental setups often miss the minimum number of repetitions to produce statistically robust conclusions. We provide empirically grounded guidelines for the number of repetitions needed to reliably compare fuzzers and offer practical insights to improve the rigor of future fuzzing research. Our work serves as a foundational step toward more proper fuzzer evaluations by offering insights into how to handle the unbearable randomness of fuzzing.

1. Introduction

Fuzzing is a popular software testing technique widely adopted both in industry and academia. Coverage-guided fuzzing, one of the most popular approaches, consists of feeding a target program with randomized inputs, mutated according to the execution path taken during the execution of an input [1]. Over the years, fuzzing has emerged as one of the most effective automated techniques for discovering security vulnerabilities. With numerous fuzzing papers published on a regular basis, the community relies on quantitative metrics–such as code coverage and bug discovery [2]–to compare new fuzzers and substantiate research claims.

The challenge. Fuzzing is inherently tied to randomness, as it relies on randomly generated inputs to explore and test a target application. However, when combined with the complexity of real-world applications, this variability makes the evaluation of fuzzers' performance challenging. To control for this variability, researchers need to repeat their evaluations across multiple trials and apply statistical tests to compare results [2], [3]. In particular, when comparing fuzzers, statistical tests can be used to accept or reject a hypothesis and determine how a fuzzer compares to the others. However, these tests do not tell if enough trials have been performed to mitigate fluctuations in the results or if the fuzzers are distinguishable with a reasonable number of trials at all. To address this, we must study the very nature of randomness in fuzzing. Unfortunately, despite its importance, this aspect has received surprisingly little attention in prior work.

The Approach. In this paper, we begin by quantifying both the causes and extent of variability in fuzzing campaigns. This analysis lays the groundwork for understanding the inherent nature of randomness in fuzzing. Through our analysis, we identify and discuss various sources of indeterminism, including the target application itself (because of time-sensitive code or the use of particular system calls), the fuzzer (through the generation of random mutations or its sensitivity to the target execution time), and the underlying hardware and software environment (e.g., noisy cloud environment, process scheduling, cache misses). We designed different experiments to isolate the effects of each of these factors on fuzzer metrics. In addition, we discuss how the variability of results could be reduced in a real-world scenario.

We also study how variability impacts the interpretation of coverage metrics in fuzzing evaluations. In particular, we assess whether the number of trials typically used in popular fuzzing benchmarks is sufficient to suppress noise and produce reliable results. Lastly, we conduct a statistical power analysis to answer the final question: how many trials are required for a statistical test to distinguish between fuzzers?

Understanding the magnitude and effect of randomness is important to let researchers understand whether experimental results reflect meaningful trends or are simply artifacts of chance. Building on recent studies [2], [3], this paper provides insights on how to properly interpret benchmark results and avoid drawing misleading conclusions from noisy experiments.

Contributions: To the best of our knowledge, this is the first study to rigorously analyze the causes and degree of randomness in fuzzing. After measuring the noise and variability over hundreds of campaigns, we use this information to distill practical recommendations for the number of trials necessary to distinguish between fuzzers' performances.

2. Background

Before tackling the problems caused by randomness during fuzz testing, we first discuss the common practices and the main challenges encountered in modern fuzzing evaluations. Next, we highlight the various sources of nondeterminism that influence a fuzzing campaign and explore how they can be observed and measured.

2.1. Evaluation of Fuzzers

Until recently, fuzzers were evaluated in an ad-hoc manner by using arbitrarily chosen datasets and inconsistent evaluation metrics. In 2018, Klees et al. [3] conducted the first systematic review of fuzzing evaluations through the analysis of 32 academic papers. The authors emphasized the need for a common and relevant set of target programs to adopt as a baseline, suggested appropriate measurement metrics, and proposed a set of configurations to adopt for fuzzing experiments. Today, it is common practice to compare new solutions with baseline approaches across a set of several targets, and to repeat the experiments multiple times. As a result, for example, an experiment comparing five fuzzers across 23 targets for 20 trials, with each campaign lasting 24 hours, would require a total of 55,200 CPU hours.

In response to the increase in the computational resources required to perform experiments and to the demand for accessible and reliable methods to conduct fuzzing evaluations that avoid common pitfalls, multiple standardized benchmark suites have gained popularity [4], [5]. In particular, FuzzBench [4] offers a unique advantage by allowing researchers to run experiments for free in the Google Cloud infrastructure, thus reducing the costs and greatly simplifying the setup and management of large-scale evaluations. Many recent papers [6]–[10], often used as baselines in the literature, rely on FuzzBench for their evaluations. Its use is also recommended in a 2024 SoK study by Schloegel et al. [2]. This work, published after the work of Klees et al., examined 150 fuzzing papers published in the previous six years, and identified a number of major additional problems, such as the lack of a statistical test for fuzzing evaluations, the poor reproducibility of experiments, and the unfair resource allocation in the reviewed experimental setups.

2.2. Sources of Non-determinism in Fuzzing

Since, at its core, fuzzing consists of testing an application by using randomly-generated inputs, randomness is intrinsic in the nature of fuzzing. In fact, all fuzzers rely on a pseudo-random number generator (PRNG) to control different parts of their operation, including the selection, generation, and mutation of test cases. As a result, multiple independent trials are typically required to draw reliable conclusions – for instance, when comparing the effectiveness of two fuzzers on a given target. Prior studies [3] [2] have highlighted this necessity, recommending that evaluations be conducted over multiple runs and supported by appropriate statistical analyses.

While this principle is generally well understood within the fuzzing community, it would be wrong to assume that fixing the seed of the PRNG (a feature offered by widely used fuzzers such as AFL++ and LIBFUZZER) is sufficient to ensure experimental reproducibility. In fact, there exist additional sources of non-determinism that can affect the outcome of a fuzzing campaign, even when the PRNG seed is fixed.

Non-Determinism in Target Applications. To reduce experimental variability—and thus limit the number of trials required to achieve statistically significant results—it is important that the target application behaves deterministically. That is, given the same input, it should consistently produce the same behavior and execution path. Some fuzzers, such as AFL++, attempt to quantify this property through a target *stability* metric, which measures the consistency of the coverage map when a particular input is executed multiple times. This metric is used by the research community as an accepted measure to quantify target-level variability [11]. Specifically, the stability metric is defined as the ratio between the number of *inconsistently* traversed edges (i.e., those visited in every run) and the total number of edges traversed by the input. As such, if one wants to measure fuzzers' performance in a deterministic condition, a recommended best practice would seem to be to use targets exhibiting 100% stability. However, whether this metric is truly reliable in evaluating target stability is an unanswered question.

Unfortunately, achieving this level of determinism is more challenging than it may initially appear. For example, in one of our experiments using the BLOATY [12] binary analysis tool—which AFL++ reports as having 100% stability, we observed that different fuzzing campaigns diverged significantly over time. That is, the sets of inputs generated and retained by the fuzzer differed substantially across runs. We will discuss this case study in our discussion Section 5.3.

The Subtle Effect of Time. If the PRNG seed is fixed and the target application exhibits fully deterministic behavior, will a fuzzing campaign always produce identical results? The answer, once again, is no.

Fuzzing experiments are typically executed across a variety of environments–including bare-metal systems, virtual machines (VMs), and containerized platforms like Docker (notably used by FuzzBench). All of these environments are susceptible to fluctuations originating from the operating system, the underlying hardware, and the environment in which the hardware is located (e.g., the room temperature) [13].

These sources of noise result in small, yet unavoidable, variations in an application's execution time. As documented in the literature [14], such variability persists even when experiments are repeated on the same physical machine under seemingly identical conditions. Although these timing differences are often minimal in absolute terms (as we will quantify in Section 4), their effect on a fuzzing process can be substantial, due to the design of modern fuzzing algorithms.

In particular, fuzzers derived from AFL implement a scheduling strategy called the power schedule [15]. This mechanism prioritizes some inputs in the queue by fuzzing them more than others, based on scores assigned to each testcase, where each score is computed from a combination of properties. A commonly overlooked factor in this scoring is the role of the target execution time:

faster testcases are often prioritized to accelerate fuzzing progress, while slower ones are penalized. As a result, even small timing variations–arising from uncontrollable external factors–can cause the fuzzer to select different testcases across runs.

As discussed previously in the context of non-deterministic coverage maps, such differences in input prioritization can lead to divergence between campaigns. Once different inputs are selected early on, the evolutionary nature of fuzzing can amplify these initial discrepancies, ultimately producing significantly different outcomes across otherwise identical experiments. We will conduct another case study and quantify this effect in Section 5.2

In summary, fuzzing campaigns are influenced by multiple sources of non-determinism—beyond the well-known randomness of input generation. Even when the PRNG seed is fixed and the target appears deterministic, subtle variations can arise from the execution environment, the internal behavior of the target program, and the fuzzer's own scheduling heuristics. These factors can cause campaigns to diverge significantly, leading to different sets of discovered inputs and coverage profiles. Consequently, drawing reliable conclusions about fuzzer effectiveness requires careful experimental design, including the use of stable targets, controlled environments, and repeated trials supported by statistical analysis. In the rest of the paper, we will measure the extent of these variations and leverage the experimental results to suggest the appropriate number of trials needed to discriminate between different fuzzers.

3. Methodology

Given the various sources of randomness that can affect a fuzzing campaign, it is primordial to consider how, or more precisely *on what*, we should assess their impact.

The most natural metric to evaluate the effectiveness of fuzzers is the number of distinct bugs discovered during a campaign. However, it is too coarse-grained and sporadic, and therefore, fails to capture small differences in the effectiveness and behavior of a fuzzer. In addition, whether bug count is a biased metric for comparing fuzzers is still actively discussed [5]. Thus, *code coverage* became the primary evaluation metric employed within the fuzzing research community. It is widely regarded as a reliable proxy for assessing the effectiveness of a fuzzer, as it quantifies the number of unique code paths exercised during a fuzzing campaign and provides insight into the stability of those paths over time. Nevertheless, code coverage exhibits a significant limitation. Over extended execution periods, coverage typically reaches a saturation point, beyond which additional increases become minimal and difficult to measure. This saturation diminishes the metric's sensitivity and discriminative power, thereby limiting its use for comparative analysis and for performance evaluation of long-running fuzzing experiments.

The third option is to assess the variability of a campaign by considering the *fuzzer throughput*, measured by counting the number of inputs generated and tested against the target application. It does not suffer from saturation, and at the same time, it serves, as demonstrated by Böhme et al. [16], as a useful proxy for code coverage (as the two are related by a logarithmic relationship). In addition, the fuzzer throughput itself is a metric people are interested in, as there are a couple of studies focusing on reducing overhead and improving performance [17]. The throughput provides enough sensitivity to observe the variability of a campaign, as the number of executions is directly proportional to the number of unique inputs generated by the fuzzer obtained through mutations.

For these reasons, in our experiments, we conduct two different measurements. First, we measure the degree of variability in the execution of a fuzzer through the lens of its throughput, to maximize the sensitivity of our measurement for smaller environmental factors. Then, we also measure the impact on code coverage to observe whether these differences are still relevant asymptotically (i.e., when coverage tends to saturate).

3.1. Metrics

Before we explain the design of our experiment, we describe the statistical estimators that we use to quantify the variability, e.g., of the coverage or total number of executions.

Since we need to compare results across different fuzzing targets, we need to adopt a dimensionless target-agnostic measure. To this end, we calculate two scale-independent metrics that make it possible for us to compare the results across different targets.

Coefficient of Variation (CV). This metric expresses the standard deviation as a fraction of the mean in a dimensionless way:

$$\boxed{\mathrm{CV} = \frac{\sigma}{\mu} \ (\%)}$$

with μ estimated using the sample mean m, and σ estimated using the sample standard deviation s. For example, a CV of $5\,\%$ means the typical run deviates from the mean by roughly five percent. In other words, 68.27% of the data is spread across the 5% range around the mean, as 68.27% is the share of data within one standard deviation of the mean in a normal distribution.

Quartile Coefficient of Dispersion (QCD). When the spread of results does not follow a normal distribution, we should avoid using the mean and standard deviation. In these cases, the interquartile range provides a more accurate metric that is less sensitive to outliers. Like in the previous case, we want a measure that is independent of the absolute scale of the target, and thus we use the *Quartile Coefficient of Dispersion*:

$$\boxed{\mathrm{QCD} = \frac{Q_3 - Q_1}{Q_3 + Q_1} \ (\%)}$$

Here, Q_1 and Q_3 are the 25^{th} and 75^{th} percentiles of all the runs. Since it uses only the middle half of the data, and does not use the mean, QCD is robust to extreme outliers, unlike the coefficient of variation, which uses the mean. Dividing by $Q_3 + Q_1$ makes it scale–free, so we can compare targets even when their average execution counts differ by orders of magnitude.

CV and QCD represent "noise-to-signal" ratios for specific quantities measured during the fuzzing process, such as coverage or throughput. These metrics provide an indication of how much variability is present relative to the observed signal. However, because they are inherently relative measures, their values cannot be interpreted in an absolute sense. In other words, a higher or lower ratio does not translate into a universal scale of "better" or "worse" performance across all scenarios. Instead, CV and QCD should be understood as context-dependent indicators, meaningful primarily when comparing experiments under similar conditions, but not as standalone measures of fuzzer quality or efficiency.

3.2. *Null Target*

To isolate the randomness originating from the target itself, we designed a special target, which we refer to as the *null-target*. All targets used in our experiment expose an entry point API for the fuzzer, named `LLVM-FuzzerTestOneInput`. Typically, developers place the code they wish to test inside this function. AFL++, in turn, repeatedly invokes this function during fuzzing, passing a pointer to the mutated input along with its length. This behavior is enabled by a piece of glue code within AFL++, as illustrated in the following code snippet.

```
while (__afl_persistent_loop(N)) {
  if (unlikely(callback(__afl_fuzz_ptr,
    *__afl_fuzz_len) == -1)) { // we disable
    this callback
    memset(__afl_area_ptr, 0, __afl_map_size);
    __afl_area_ptr[0] = 1;
  }
}
```

Through consecutive calls to the target function, the fuzzer executes the target program under test. We modified AFL++ so that the fuzzer no longer invokes this function so that no target code is executed at all. This modification allows us to completely eliminate the influence of the target itself and thus measure how variability in fuzzer performance is affected solely by external factors such as hardware or operating system-level environmental noise.

3.3. Experiments Design

As we discussed in the previous section, several different factors influence the variability of a fuzzing campaign. To evaluate how much each source contributes to the total, we performed a number of experiments by controlling three main dimensions that can affect the noise in the results:

Target: We experimented with stable targets (*stability*=100% according to AFL++), unstable targets (*stability*<100%), and a *null-target*; a test program with an empty fuzzed (`LLVMFuzzerTestOneInput`) function.

Environment: We run fuzzing campaigns both on a local server and on a cloud infrastructure.

PRNG: We performed experiments both with and without a fixed seed for the random number generator. In addition, we used the simpler AFL++ queue-like scheduler.

We combined these factors in four different experiment campaigns plus a meta-campaign that analyzes already existing data. The first two campaigns are designed to assess the environmental noise. They include identical experiments performed by using the FuzzBench fuzzing framework: one deployed on a local machine (Campaign **C-I**) and one on the Google Cloud Platform Infrastructure (Campaign **C-II**). In the first case, the entire configuration is under our control, and the hardware remains the same for the entire duration of the experiments. For this experiment, we run the fuzzer on a VM with Ubuntu 20.04.6 LTS (the only VMs on the server) with 96 cores (Intel(R) Xeon(R) Platinum 8260M no Hyper-Threading), and 512GiB of RAM The FuzzBench's cloud installation instead abstracts away all the infrastructure details, making it unclear whether all fuzzer instances run on the exact same hardware or even on comparable nodes. In addition, the cloud itself (due to the presence of multiple concurrent VMs) introduces another source of nondeterminism and noise in the results.

In both campaigns, we only tested our `null` target: since this program only provides an empty function to be fuzzed, it minimizes any potential differences in execution time and coverage feedback that can influence the execution of the fuzzer, isolating only the environmental noise for our observation

Thanks to **C-I** and **C-II**, we can establish the level of background noise that researchers might expect when running a fuzzing campaign. We then compare this baseline with the actual variability we can observe in real experiments by using two additional experiment campaigns.

The additional two fuzzers setups are identical, except for one aspect. In both cases, we tested ten stable and ten unstable target applications on the FuzzBench cloud infrastructure. However, in the first (Campaign **C-III**) we disable the randomness of the fuzzer by fixing the seed of the PRNG algorithm (by using the `-s` option of AFL++) and by using a deterministic scheduling policy that picks corpus items sequentially in a queue-like way (by using the `-Z` option). This is the best possible way users can configure AFL++ without further compromising its core algorithm. In the second campaign (**C-IV**) we use instead a standard configuration as used by the majority of researchers, which is the default config of FuzzBench, where both the fuzzer and targets are executed without any modifications.

For Campaign **C-III**, we performed another set of fuzzbench runs for 8 selected *stable* targets on local FuzzBench (which we call Campaign **C-IIIL**). This campaign shares the same setup as Campaign **C-III** but it run on a local machine instead of on the remote FuzzBench cloud infrastructure. With this experimental setup we can answer how the results from remote experiments are more or less unstable than the the ones from local experiments. The comparison between Campaign **C-III** and Campaign **C-IIIL** makes it possible for us to compare the the environmental noises on real targets.

Impact on Coverage. For the four campaigns described so far (**C-I** to **C-IV**), we run the fuzzer in different scenarios and we measure the variability in terms of changes on the fuzzer throughput, i.e., on the number of inputs per second generated on average over a period of 23 hours. For our final experiment, we shift our attention to the impact of this variability on the final coverage obtained at the end of the fuzzing campaign.

TABLE 1: Throughput variability of the `null` target in terms of the total number of executions over 23h in millions.

	N	Min	Max	Mean	CV%	QCD%
Campaign C-I	100	536	565	587	1.78	1.23
Campaign C-II	65	1966	2120	2255	3.57	3.05

For this meta-campaign (**C-V**), we accumulated the fuzzer coverage data across nine independent experiments conducted on FuzzBench in the past that evaluated a set of 11 different fuzzers with the most standard setups, running on 22 different targets. Importantly, all these nine experiments are conducted at different times, spanning around one month. We confirmed that they use the same version of the fuzzers and the targets, making the results comparable to each other. Each of these fuzzers has no further modifications to control the randomness. This will reflect a more realistic situation, as in reality nobody would want to evaluate fuzzers in a special condition described the experiment of campaign **C-III** which controlls sources of randomnesss.

4. Experiments Results

4.1. Environmental Noise in Isolation

In the first two campaigns (**C-I** and **C-II**), we measure the variability in the fuzzer throughput when all non-deterministic aspects of the fuzzer are removed (by using the `null` target and fixing the randomness seed). In other words, in these two experiments, the environmental noise is specifically isolated and quantified.

For **C-I**, we repeated 100 fuzzing experiments of 23 hours each on FuzzBench running within our local server infrastructure. We then computed the average throughput and observed how it changed from one test to another.

Table 1 presents the results of the first experiment. The average total number of executions ranged from 536 to 587 million. A Shapiro-Wilk test [18] with $\alpha = 0.05$ showed a P-value of 0.68, indicating that the distribution of the results is compatible with a Gaussian distribution. This outcome aligns with expectations: the results derive from a stochastic variable influenced by the cumulative effects of multiple independent noise sources, such as cache misses, CPU temperature variations, and OS scheduler impacts.

We repeated the same experiment for **C-II** in the FuzzBench instance hosted on Google Cloud. Due to the instability of the setup, which is not under our control, some runs were interrupted before reaching the planned 23 hours. As a result, we only included the fully completed tests, totaling 65 fuzzing experiments of 23 hours each.

As shown in Table 1, the raw throughput of the remote infrastructure is approximately 3.7x higher than on the local deployment. However, the variability scaled with the mean (CV) has doubled compared to the controlled environment, and the distribution no longer fits a Gaussian curve (P-value = 0.03 and several small peaks appear when plotted). In fact, studies have shown that the performance of CPU-bound applications often follows multimodal distributions, especially when deployed in the cloud [19].

Takeaways: Without considering the fuzzer and the target application, the hardware-software environment alone is responsible for small-but-relevant variations in the throughput of a fuzzing campaign. This noise is nicely distributed when experiments are conducted in a controlled environment, but it becomes twice as large and more unpredictable when a cloud infrastructure is employed.

4.2. Enviromental Noise in Practice

Lastly, we analyze the results from Campaign **C-IIIL**. This setup measures the environmental noise in practical setups. Table 4 compares the stable-target experiments from Campaign **C-IIIL** and Campaign **C-III**.

Campaign **C-IIIL** and Campaign **C-III** are identical in setup, with the only difference being that Campaign **C-IIIL** was executed on a local machine, whereas Campaign **C-III** was conducted on the remote FuzzBench infrastructure. The table shows no consistent trend indicating whether remote experiments are more or less stable than local ones. This suggests that environmental noise is a relatively minor factor influencing the fuzzer's randomness, and that running experiments remotely does not inherently lead to less stable evaluation results.

The remaining factors contributing to the fuzzer's randomness will be examined in the next section.

4.3. Fuzzer and Target Noise

In the previous experiments, we learned that the environmental noise is small but not negligible, and one should expect the fuzzer's throughput to fluctuate by a few percentage points around the mean. But what happens when we substitute the `null` target with a real program?

In the presence of real targets, there are three more factors that contribute to the final result: non-determinism in the target application possibly affecting fuzzer coverage, randomness of the fuzzing process (PRNG), and the sensitivity of the fuzzer scheduler to fluctuations in the target execution time. To assess their amplitude, we can look at the results of remote Campaigns **C-III** ("deterministic" fuzzer) and **C-IV** (standard fuzzer), whose results are reported in Table 2 for stable targets (according to AFL++) and Table 3 for unstable targets.

Each experiment lasted for 23 hours and was repeated for 20 trials. We then asked the remote FuzzBench instance to repeat everything 5 times, thus potentially providing a total of 100 trials for each target application. Unfortunately, due to the instability of the FuzzBench platform, not all trials were successfully executed for the requested 23 hours (and, sometimes, some experiments were executed for more than 20 trials). Therefore, we collected the artifacts of the fuzzer instances that lasted at least 23 hours and report their number in the N column in the two tables.

Even at a first glance, we can observe that there is no clear difference between the two tables. In other words, while researchers might want to refrain from using

TABLE 2: Throughput variability of the Stable Targets, in terms of the total number executions over 23h. Min, Max, and Mean are given in millions.

	Campaign C-III						Campaign C-IV					
Stable Target	N	Min	Max	Mean	CV%	QCD%	N	Min	Max	Mean	CV%	QCD%
`bloaty_fuzz_target`	102	68	125	100	12.64	8.57	60	45	132	87	27.14	26.22
`jsoncpp_jsoncpp_-fuzzer`	100	43	526	271	42.03	27.73	52	214	790	520	25.80	11.56
`lcms_cms_-transform_fuzzer`	100	4	195	90	44.92	33.38	61	16	379	209	36.82	19.93
`libjpeg-turbo_-libjpeg_turbo_-fuzzer`	100	276	527	450	9.44	4.26	60	166	754	528	23.37	13.99
`libpng_libpng_-read_fuzzer`	100	440	811	653	10.81	7.24	63	829	1132	993	6.66	4.92
`mbedtls_fuzz_-dtlsclient`	101	224	499	443	7.86	1.37	59	400	520	464	4.18	1.87
`openssl_x509`	100	965	1129	1072	2.91	1.70	63	1048	1319	1137	4.59	2.60
`vorbis_decode_-fuzzer`	101	2	17	8	41.79	34.34	62	27	86	59	26.15	20.76
`woff2_convert_-woff2ttf_fuzzer`	101	138	591	380	30.87	23.60	55	25	95	51	32.01	23.30
`zlib_zlib_-uncompress_fuzzer`	101	267	1655	1038	37.24	33.28	58	990	2022	1453	15.46	11.18

TABLE 3: Throughput variability of the Unstable Targets, in terms of the total number of executions over 23h. Min, Max, and Mean are given in millions.

	Campaign C-III						Campaign C-IV					
Unstable Target	N	Min	Max	Mean	CV%	QCD%	N	Min	Max	Mean	CV%	QCD%
`curl_curl_-fuzzer_http`	101	667	978	900	5.78	3.27	50	871	1073	946	5.00	3.97
`freetype2_-ftfuzzer`	100	230	425	323	10.09	6.78	58	482	676	590	6.44	3.63
`harfbuzz_-hb-shape-fuzzer`	100	156	213	188	4.99	2.74	58	74	172	129	17.29	12.78
`libpcap_fuzz_both`	100	137	492	394	22.27	12.98	61	272	503	400	12.18	7.32
`libxml2_xml`	101	182	243	209	6.45	4.75	67	57	182	114	23.79	15.97
`libxslt_xpath`	102	434	1203	805	20.72	15.28	64	1016	1625	1306	9.19	6.10
`openthread_-ot-ip6-send-fuzzer`	101	114	253	228	9.89	0.86	53	225	269	235	3.57	0.80
`proj4_proj_crs_-to_crs_fuzzer`	80	91	158	118	10.21	4.76	68	138	235	184	10.56	4.79
`re2_fuzzer`	101	11	62	27	34.67	21.66	54	43	210	129	28.61	19.94
`sqlite3_ossfuzz`	101	47	640	70	106.77	4.70	58	77	1606	252	130.25	7.34

unstable targets for better reproducibility and consistency in comparative experiments, it seems like the variability of the results is *not* higher for those applications (with the sole exception of `sqlite3` being extremely unstable). While this might initially seem counterintuitive, the phenomenon can be explained. We will discuss in the Section 5.3, the measure of stability is quite fragile and therefore even targets that are reported as 100% stable by AFL++ might in fact exhibit non-deterministic differences in their behavior.

A second result we can draw from the two tables is the significant difference in variability observed between experiments **C-III** and **C-IV**. Following [2], [3], we conducted a Mann-Whitney U test [20] with $\alpha = 0.05$ for each target in the two tables, comparing the throughput data from the **C-III** experiments (fixed PRNG) with that from **C-IV**. We chose the Mann-Whitney U test because it is a robust non-parametric test, and it does not assume normality of the data, a condition frequently violated in fuzzing experiments [2].

For all targets except `libpcap_fuzz_both` and `openthread_ot-ip6-send-fuzzer`, the test confirms that the two distributions are significantly different, indicating that fixing the PRNG has a notable impact on the variability of the throughput. However, a more subtle aspect emerges when we consider the coefficient of variation (CV) as a random variable dependent on the target and fuzzer configuration. When analyzing all four combinations, PRNG on/off and stable/unstable targets, the Mann-Whitney U test consistently fails to show that

TABLE 4: Throughput variability of the Stable Targets, in terms of the total number executions over 23h. Min, Max, and Mean are given in millions.

	Campaign C-IIIL						Campaign C-III					
Stable Target	N	Min	Max	Mean	CV%	QCD%	N	Min	Max	Mean	CV%	QCD%
`bloaty_fuzz_target`	100	28	44	37	6.40	3.11	102	68	125	100	12.64	8.57
`lcms_cms_-transform_fuzzer`	100	4	178	75	45.49	31.40	100	43	526	271	42.03	27.73
`libpng_libpng_-read_fuzzer`	100	226	407	318	9.92	6.00	100	4	195	90	44.92	33.38
`openssl_x509`	100	310	426	365	5.36	2.66	100	965	1129	1072	2.91	1.70
`vorbis_decode_-fuzzer`	100	4	61	16	62.85	40.07	101	2	17	8	41.79	34.34
`woff2_convert_-woff2ttf_fuzzer`	100	53	583	198	73.95	57.79	101	138	591	380	30.87	23.60
`zlib_zlib_-uncompress_fuzzer`	100	336	801	593	16.83	10.78	101	267	1655	1038	37.24	33.28

the CVs are statistically different across the cases.

In other words, the throughput of a non-random fuzzer running a stable target varies as much as the throughput of a random fuzzer with a non-stable target. Thus, there must be another major factor that dominates the other and is responsible for most of the variability we observed in our experiments.

Manually inspecting the list of test cases scheduled by the fuzzer revealed significant variation in the entries selected from one trial to another. Surprisingly, even in the deterministic experiments (fixed PRNG and stable targets), the scheduled test cases rapidly diverged. This could be due to two factors:

- Tiny changes in the execution time of each test case, introduced by the environmental noise, were amplified by the scheduler, and as a result, it will change the testcase picked by the fuzzer across different runs.
- Small internal changes in the execution of the target application led to tiny changes in the coverage (discussed further in Section 5.3). Those changes were then amplified by the input selection algorithm of the fuzzer, which only keeps inputs that increase the coverage map and discards the others. This leads to some inputs being preserved in some runs but not in others.

Takeaways: The throughput of a fuzzer on a given target application can vary dramatically between runs – in some extreme cases, by as much as 20.9× (in the case of `sqlite3`). This variability remains high even on stable targets and despite fixing the PRNG seed. The underlying reason is that, in this regard, the behavior of modern fuzzers resembles that of a **chaotic system**: tiny perturbations caused by the environmental noise – such as slight differences in coverage feedback or in the target execution time – can cause the fuzzer to diverge significantly in the sequence of inputs it generates.

4.4. Variability in the Fuzzer's Coverage

So far, we have learned that several factors, some of which cannot be controlled by the analyst, can cause the fuzzer to diverge in the number and nature of test cases generated over a campaign. The next big question we want to answer is whether these variations are reflected in the final coverage obtained by the fuzzer.

Unfortunately, in 2025, it is not possible anymore to generate coverage reports from FuzzBench experiments. This feature, however, was available in the past. Therefore, we inspected historical data and searched for different campaigns that researchers had conducted by using the exact same versions of fuzzers and targets under test, permitting us to perform a meta-analysis on existing results (Campaign **C-V**). Luckily, we identified nine campaigns, executed between mid-April to early June of 2021 that compared the same set of 11 fuzzers (identical versions of AFL [1], AFLFAST [15], AFL++ [6], AFLSMART [21], ECLIPSER [22], ENTROPIC [23], FAIRFUZZ [24], HONGGFUZZ [25], LAF-INTEL [26], LIBFUZZER [27], MOPT [28]) on the same 22 targets[1].

While each experiment should have included 20 trials, as we discussed before, it was not always possible to retrieve all results from FuzzBench, as some were not executed in their entirety. Despite this limitation, the number of completed and available data points gathered over all experiments for each combination of fuzzers and targets varied between 106 and 154. We gathered all the raw data from the nine experiments, including their coverage maps, and report different metrics in Table 5. Due to the space limitation, we only present results for five of the eleven fuzzers, while the full results are available in the artifact file. We present the mean region coverage [29] (the default coverage of FuzzBench at the time of the experiment in 2021) along with its CV%. All 9 experiments use the same targets and fuzzer pair, but since FuzzBench is updated after 2021, the target name in Table 5 does not necessarily match those in the previous throughput experiments conducted in 2025.

The first interesting observation is that the coverage variability depends on the target rather than the fuzzer. Namely, there are targets that always present a high (or low) variance, but there are no fuzzers whose coverage variance is constantly high (or low). For example, targets like `openssl_x509`, `jsoncpp_jsoncpp_fuzzer`,

1. These full list experiments is available at https://github.com/BBB-paper/BBB-artifact

TABLE 5: Mean coverage with its Coefficient of Variation (CV) for each target and fuzzer formatted as [$coverage \pm CV\%$]

Benchmark	AFL	AFLFast	AFL++	Honggfuzz	LibFuzzer
bloaty_fuzz_target	8358 ± 4.45%	8066 ± 5.23%	8441 ± 4.31%	8019 ± 2.99%	6948 ± 5.58%
curl_curl_fuzzer_http	17624 ± 0.36%	17449 ± 0.58%	17960 ± 0.56%	17965 ± 0.53%	15991 ± 5.25%
freetype2-2017	20623 ± 1.29%	20108 ± 2.50%	28013 ± 3.91%	28662 ± 2.71%	19506 ± 5.76%
harfbuzz-1.3.2	8402 ± 1.06%	8345 ± 1.06%	8580 ± 2.36%	8605 ± 1.08%	8335 ± 1.23%
jsoncpp_jsoncpp_fuzzer	638 ± 0.08%	638 ± 0.08%	638 ± 0.08%	640 ± 0.05%	641 ± 0.00%
lcms-2017-03-21	2521 ± 8.30%	1898 ± 29.40%	3160 ± 12.71%	3108 ± 14.22%	3271 ± 4.09%
libjpeg-turbo-07-2017	3733 ± 3.19%	3650 ± 4.32%	3804 ± 0.51%	3692 ± 3.81%	3595 ± 8.08%
libpcap_fuzz_both	102 ± 63.12%	109 ± 87.38%	4443 ± 4.70%	4176 ± 5.68%	3530 ± 3.55%
libpng-1.2.56	1512 ± 0.43%	1510 ± 0.04%	2083 ± 0.56%	2150 ± 1.18%	1990 ± 1.07%
libxml2-v2.9.2	10228 ± 13.82%	6929 ± 8.36%	12278 ± 6.34%	11180 ± 10.84%	6922 ± 8.32%
libxslt_xpath	18858 ± 0.37%	18676 ± 0.34%	18981 ± 0.62%	18892 ± 0.72%	15788 ± 0.74%
mbedtls_fuzz_dtlsclient	8572 ± 11.11%	8105 ± 0.45%	8305 ± 4.41%	8174 ± 1.58%	7990 ± 2.73%
openssl_x509	13778 ± 0.02%	13767 ± 0.09%	13773 ± 0.06%	13739 ± 0.09%	13747 ± 0.05%
openthread-2019-12-23	5795 ± 2.58%	5438 ± 5.00%	5844 ± 0.38%	5835 ± 0.73%	5895 ± 0.17%
php_php-fuzz-parser	43435 ± 0.24%	43275 ± 0.39%	43761 ± 0.39%	43825 ± 0.31%	42173 ± 0.51%
proj4-2017-08-14	4541 ± 14.51%	4046 ± 19.54%	5617 ± 2.10%	7670 ± 1.79%	7695 ± 1.12%
re2-2014-12-09	3526 ± 0.54%	3519 ± 0.55%	3530 ± 0.52%	3512 ± 0.40%	3516 ± 0.61%
sqlite3_ossfuzz	35014 ± 1.62%	34620 ± 1.30%	36657 ± 0.65%	30223 ± 2.04%	23815 ± 4.80%
systemd_fuzz-link-parser	640 ± 0.00%	640 ± 0.03%	640 ± 0.00%	639 ± 0.00%	624 ± 2.97%
vorbis-2017-12-11	2166 ± 0.22%	2163 ± 0.28%	2168 ± 0.14%	2152 ± 0.27%	1975 ± 8.18%
woff2-2016-05-06	1864 ± 0.94%	1828 ± 0.63%	1865 ± 0.96%	1891 ± 1.11%	1684 ± 3.35%
zlib_zlib_uncompress_fuzzer	964 ± 0.77%	949 ± 1.20%	965 ± 0.87%	965 ± 0.70%	974 ± 1.02%

libxslt_xpath, php_php-fuzz-parser, re2-2014-12-09, and systemd_fuzz-link-parser always have the coefficient of variance under 1% across all the fuzzers. In addition, in most of these cases, all six fuzzers reached almost exactly the same coverage – suggesting that they all saturated after exploring the same part of the target.

Most of the other targets present a coefficient of variability (CV) below 10%. As expected, these values are lower than the variations we measured in Section 4.3 on the fuzzer throughput due to the logarithmic relation between the throughput and the coverage [30]. While this can be explained by the fact that coverage is a less sensitive metric compared to the throughput, it does not mean that this variability has a smaller impact on the interpretation of a fuzzing campaign. We can say coverage is a relatively more stable metric; however, it is not consistent to the eyes of the interpreter.

For instance, on curl_curl_fuzzer_http HONGGFUZZ performs better than AFL++ when the two were compared on 25 May 2021 (Day 6 on 1). However, when the exact same fuzzers were compared again on 29 May 2021 (Day 7 on 1), AFL++ outperformed HONGGFUZZ. Since researchers often perform a single batch of 20-runs experiments on FuzzBench, this shows that several results reported in the past might be completely overturned if experiments are repeated at another point in time. In essence, one should be careful in concluding the superiority of a fuzzer over another. We will analyze this problem in terms of "fuzzer rankings" in the next section.

Takeaways: Our analysis shows that the coverage is a more stable metric, and less prone to noise than throughput. Still, for experiments performed on the cloud, one can expect coefficient of variations of 5-to 10% for many targets. However, even small CVs may have a consistent impact in the comparison of different fuzzers, making it difficult and error-prone to conclude that one fuzzer is better than another without performing a very large number of experiments.

4.5. Impact on Rankings

The original goal of these nine experiments was to compare different fuzzers and determine which of them performed better on the selected targets. Unfortunately, this conclusion is also subject to noise.

Ranking the fuzzers based on their median coverage is commonly done by the fuzzing community to prove the superiority of a new technique or fuzzer. Is this leading to consistent rankings? Sadly, as Figure 1 shows, the fuzzer rankings can change completely from one experiment to another, due to the fluctuations in the coverage measurements. In fact, among the 22 targets, there was only one (libpng-1.2.56) for which the fuzzer ranking remained unchanged across different experiments. For all the other targets, the amplitude of variability between different experiments was sufficient to cause changes in the ranking. In some cases, such as libpcap_fuzz_both and freetype2-2017, the changes only affect

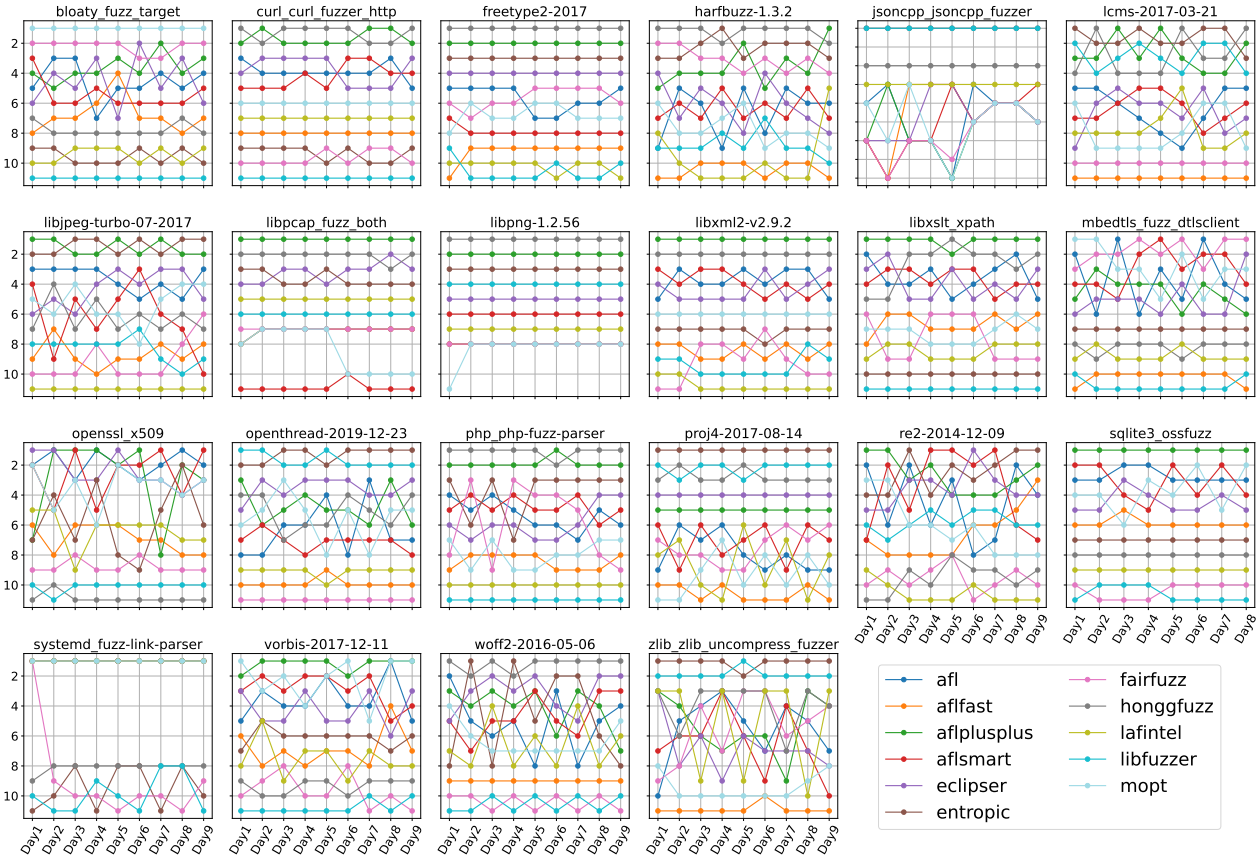

Figure 1: Fuzzer Rankings for Campaign C-V according on their median coverage.

a few fuzzers. However, for others, the changes affected the fuzzer's rankings severely. For instance, the second-best fuzzer on a given experiment ranked in ninth position when the experiment was repeated on a different day.

Takeaways: Fuzzer rankings should be presented with caution, since they are particularly fragile and largely affected by noise and variability in the coverage results.

5. Discussion

5.1. *Repetita Iuvant*

As noted in a recent meta-analysis of fuzzing literature by *Schloegel et al.* [2], statistical tests can significantly enhance the validation and reproducibility of fuzzing results. The meta-analysis reports that that over 60% of published papers entirely omit statistical analysis and that *"... 55% of the papers use fewer than 10 trials in at least one experiment"*. The authors [2] also observe that prior works, such as one from *Klee et al.* [3] often fail to rigorously define thresholds or even ranges for the number of repetitions to perform. Instead, they rely on conventional practices, for instance, the default setting of 20 repetitions commonly used for remote FuzzBench instances.

However, when a test indicates non-significant results it remains unclear whether this is due to an actual lack of difference or simply an insufficient number of trials.

Our analysis in this section provides an approximate formula for determining the number of tests needed to demonstrate statistically significant effects in hypothesis testing. This is a crucial aspect of planning meaningful fuzzing experiments.

In an ideal scenario, a baseline fuzzer's performance metric (e.g., throughput or coverage) follows a distribution (not strictly normal) with mean μ_0 and variance σ_0 across multiple runs, depending primarily on the target and environment. In this case, one can estimate the minimum number of runs needed to detect a statistically significant difference, if it exists, between the baseline's mean and that of another fuzzer, using a Mann-Whitney U test [20], [31]. The estimated minimum sample size depends on a chosen confidence level α (percentage of false positives that we accept) and statistical power $1-\beta$ (percentage of false negatives) and remains valid for $n \geq 10$. The minimum number of runs can be computed with:

$$n = \frac{(Z_{1-\alpha} + Z_{1-\beta})^2}{6\left(\Phi\left(\frac{\delta}{\sqrt{2}\sigma_0}\right) - \frac{1}{2}\right)^2} \approx \frac{1.80}{\left(\Phi\left(\frac{P}{\sqrt{2}CV}\right) - \frac{1}{2}\right)^2} \quad (1)$$

where $\delta = |\mu_A - \mu_0|$, with $\mu_A > \mu_0$ represents the estimated difference between the mean value of the measured value of the fuzzer under evaluation and that of the baseline. The terms $Z_{1-\alpha}$ and $Z_{1-\beta}$ denote the quantiles of the standard normal distribution (corresponding to α an $1-\beta$, respectively) and Φ is the CDF of the standard Gaussian. Noting that $\frac{\sigma_0}{\mu_0}$ represent the coefficient of variation CV of the baseline fuzzer, introduced in Section 3, which is a measure of the noise introduced by

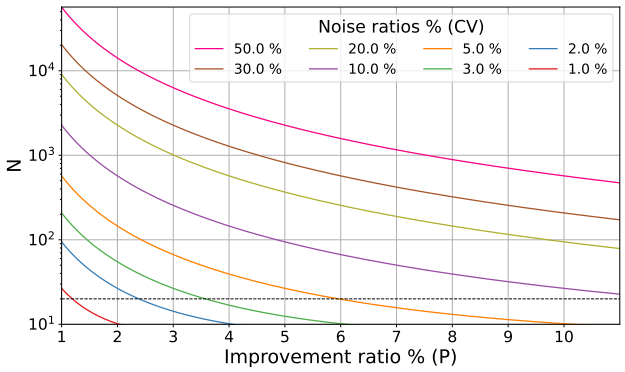

Figure 2: Number of repetitions required as a function of the percentage ratio between the means of the two fuzzers P, for different values of noise level (CV).

the hardware and software environment and the specific characteristic of the target, and choosing $\alpha = 0.05$ and $1 - \beta = 0.95$, the expression can be approximated as shown on the right side of Formula 1 with P as the percentage of improvement over the baseline, which is a more intuitive way to express the difference between the means.

In Figure 2, we plot the number of repetitions required as a function of the percentage ratio between the means of the two fuzzers P, for different noise level values (CV). The plot is limited to values $n \geq 10$ due to the limits of the approximation of the estimate of n introduced in [20], [31].

Minimum Detectable Effect (MDE). The Formula 1 and its representation in Figure 2 provide a means to estimate the number of repetitions during experiments planning. Conversely, it allows estimating the minimum mean ratio that to statistically prove experiment significance given a certain noise and a chosen number of repetitions. This allows us, in particular, to determine the minimum statistically significant improvement in throughput that can be detected for standard FuzzBench targets in a reduced-noise environment, when compared to AFL++, given a fixed number of repetitions.

By using Formula 1 and using the noise level measured in our 100 repetitions experiments (Table 2 and Table 3), we compute the minimum statistically significant improvement in throughput that can be detected under these conditions. Among stable targets, the minimum improvement that can be statistically validated with only 20 repetitions is 3.47% for `openssl_x509`, while for unstable targets it is 5.95% for `harfbuzz_hb-shape-fuzzer`. On the other hand, for other targets like `lcms_cms_transform_fuzzer`, 20 repetitions can only distinguish differences in performance if greater than 53.3%.

We can also compute the absolute minimum MDE, only based on the environmental noise we measured on the `null` target. In this case, 20 repetitions on a cloud FuzzBench deployment can statistically discern improvements larger than 4.25% (2.13% on a local installation).

Takeaways: Even on an empty application that eliminates all target- and fuzzing-related noise, 20 runs on the FuzzBench can only statistically detect improvements larger than 5%. For real targets, this number can increase, even by an order of magnitude.

Minimum Detectable Difference in Coverage. A similar analysis can also be conducted for fuzzing coverage. In this case, we perform a meta-analysis of the coverage data collected in the **C-V** experiments. For each target across the 9 collected experiments, we compute the number of repetitions required to detect a statistically significant difference between every pair of fuzzers, and we assess whether the number of repetitions actually performed is sufficient for that purpose. To our knowledge, this is the first time such an analysis has been reported in the literature.

To perform this analysis, we use the general formula [20], [31] from which we previously derived Equation 1, applicable when the two experiments being compared have different numbers of repetitions and variances. In Figure 3, we present four illustrative examples that exhibit distinct and noteworthy behaviors. In each figure, each cell reports the estimated number of repetitions, computed using the formula, required to detect a statistically significant difference between two fuzzers.

Cells are color-coded for clarity: green indicates that the required number of repetitions is both sufficient to detect a possible statistically significant difference and below 20, the threshold that we have already introduced before. Note that values below 7 are not present, as the approximation used in the formula becomes invalid in that range. Yellow cells represent cases where the number of repetitions performed is sufficient but exceeds 20. Red cells indicate cases where the performed number of repetitions is insufficient to demonstrate a statistically significant difference.

Additionally, the fuzzers are grouped into three blocks based on their lineage: the first and largest group includes fuzzers derived from AFL, the second from LIBFUZZER, and the third contains only HONGGFUZZ.

Specifically, in Figure 3, we present results for targets `libpng-1.2.56`, `freetype2-201`, `openssl_x509` and `mbedtls_fuzz_dtlsclient`. For targets like `libpng-1.2.56` 20 repetitions are generally sufficient to detect statistically significant differences not only between fuzzers from different families but also within the same family. Only a few cases require more repetitions. For other targets, such as `freetype2-201` and `openssl_x509`, instead, in general 20 repetitions are sufficient to detect significant differences between fuzzer of different families while for intra-family comparison are not sufficient anymore. In contrast, for `mbedtls_fuzz_dtlsclient`, the number of repetitions required is almost always greater than 20 and, in many cases, far exceeds the number actually used in the experiments. This applies not only to comparisons between fuzzers from the same family, which may naturally behave similarly and thus be harder to distinguish, but also to comparisons across different families.

For targets like `mbedtls_fuzz_dtlsclient`, the large number of repetitions required to differentiate fuzzers may be due to at least two, not mutually exclusive,

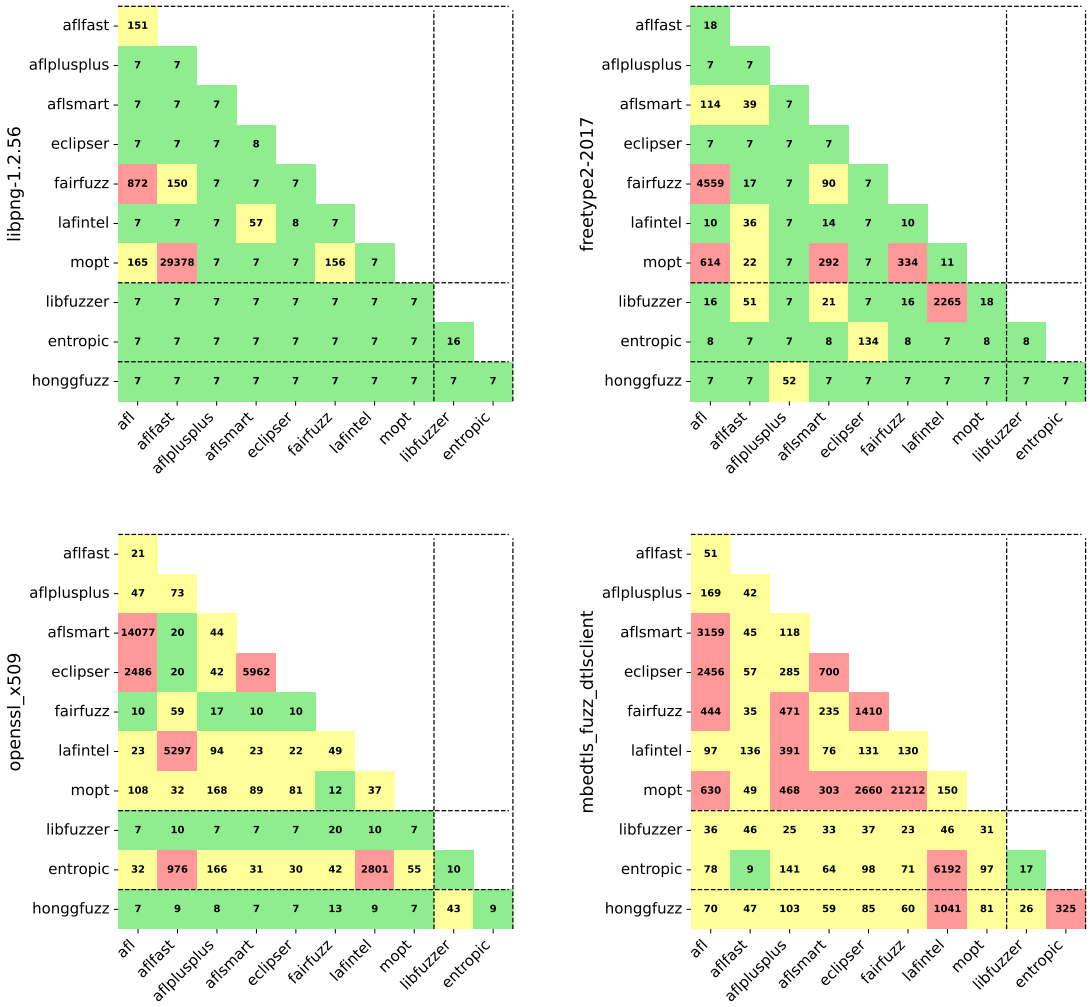

Figure 3: Number of repetitions required to detect a statistically significant difference in converage between fuzzers for four targets. Dashed lines divide fuzzers of different families.

factors: either the fuzzers, particularly those from the same family, exhibit very similar behavior on the target using same algorithms and strategies, or the target itself is either highly stable or highly unstable, causing all fuzzers to explore the input space in a similar way, or to hit the same limited set of branches.

Across 20 fuzzing targets, each tested with 11 different fuzzers, we find that for 54.63% of all fuzzer pairs, a statistically significant effect, if present, can be detected with just 20 repetitions. In 30% of the cases, the number of repetitions performed on the platform is sufficient but exceeds 20, while in the remaining 15%, the number of repetitions is insufficient to detect a significant difference.

Notably, for 17 out of 20 targets, at least one pair of fuzzers from different families exists that cannot be statistically distinguished with only 20 repetitions. Furthermore, for every single target, there is at least one pair of fuzzers, both belonging to the AFL family, for which 20 repetitions are also insufficient to establish a statistically significant difference.

5.2. Evolution of CV Over Time

So far, we have only discussed the variability of throughput and coverage as observed at the end of a 23h campaign. However, it is still unclear whether the input values generated by the fuzzer start diverging immediately or only after a certain period of time. Figure 4 shows how the coefficient of variance CV changed over time for two of our targets (Campaign **C-III**). The influence of randomness becomes pronounced only after an initial period, indicating that outcomes collected early in a fuzzing campaign tend to exhibit greater stability.

From the graph, we can see that the variability is considerably lower in the first 15 minutes, but then rapidly increases and reaches much higher values within two hours. This could benefit those studies that do not require long-term experiments, as the lower variability observed over short runs can make it easier to distinguish even small effect sizes. For example, Jiradet et al. [32] presented a benchmark that uses a shorter amount of experiment time, but still works as a good approximation of the FuzzBench benchmarking result.

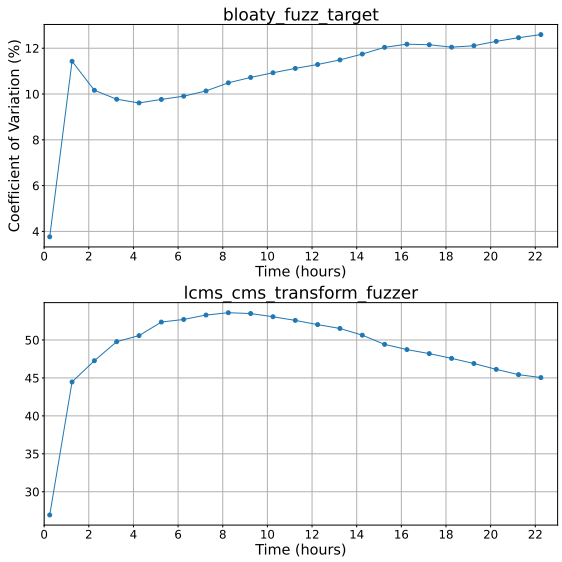

Figure 4: Coefficient of variation over time for two targets. Data points correspond to measurements taken at the 15th minute of each hour.

Takeaways: The variance of a fuzzer throughput is amplified over time. If one does not require long-term evaluations, the effect of randomness is considerably reduced.

5.3. Noise Amplification: a Case Study

In Section 4.3 we show that even when the seed of the PRNG is fixed and the target is stable, we still observe large variations in throughput of a fuzzer across different campaigns. To better understand the reasons behind this phenomenon, we manually investigated the internal details of several experiments. Due to space limitations, we discuss here one example, related to the `bloaty_fuzz_target`.

The lists of generated test cases showed differences after just a few seconds of fuzzing. In particular, we pinpointed the origin of the divergence to a mutational stage called *inference stage*. Essentially, this stage mutates and runs the test case taken from the corpus. Although this is quite a commonplace routine for fuzzers, after AFL++ executes the mutated input, it computes a hash of the coverage map and adjusts its mutation strategy depending on the result.

In our experiments, we observed that the exact same input was deemed interesting (and thus retained in the corpus) in some runs, but it was discarded as non-interesting in others. Since inputs are considered interesting when they induce changes in the coverage map, this inconsistency indicates some edges were traversed in one run but not in the others. A tedious manual investigation revealed that the discrepancy was due to a `futex()` system call that occasionally failed, leading to small differences in the internal execution paths–even though the external behavior of the program remained unchanged.

This highlights the fact that although BLOATY_FUZZ_TARGET was reported as 100% stable by AFL++, the target showed small internal non-deterministic behaviors that were not externally observable.

Takeaways: The *stability* metric can be imprecise, as it is only calculated using the saved corpus test cases and not all the mutated test cases tried and executed against the target. Moreover, modern fuzzers are not robust against small fluctuations in the internal coverage, which can cause large differences in the list of generated test cases.

5.4. FuzzBench Status in 2025

Our throughput analysis, including **C-III**, was conducted on the FuzzBench online benchmarking infrastructure in 2025. In contrast, the coverage analysis presented in Section 4.4 relies on archived FuzzBench data from 2021. Unfortunately, by 2025, it was no longer possible to reliably obtain FuzzBench coverage reports. From the experiment runs, we were still able to collect fuzzer logs to measure throughput, but computing coverage was not feasible. Therefore, for the throughput analysis, we executed a new set of experiments in 2025, while for the coverage analysis, we relied on historical FuzzBench data from four years earlier.

5.5. Consistency in the Number of Runs

In both our throughput and coverage analyses, the number of runs for each fuzzer-target setup is not consistent, except for the local experiment in **C-I**. For instance, in Campaigns **C-II**, **C-III**, and **C-IV**, we requested 100 runs of 23 hours each, but in practice, the number of completed runs sometimes differed from this target. This variation arises from the way the FuzzBench infrastructure operates. FuzzBench experiments run on preemptible virtual machines (VMs), which can be terminated before reaching the full 23-hour runtime. To compensate, the infrastructure may launch additional instances beyond the requested number to ensure sufficient data collection. Conversely, in some cases, many VMs fail to reach the 23-hour mark, resulting in fewer than 100 runs being completed.

It might seem more consistent to standardize the number of runs across all experiments, for example, using 50 runs for every campaign, since all experiments include at least that many. However, as our primary goal is to maximize the accuracy of our analysis, we chose to use all available runs rather than subsampling to a fixed count.

5.6. Practical Suggestions

The results in Section 5.1 demonstrate that with a limited number of runs, it is often difficult to determine whether one fuzzer truly outperforms another. In some cases, an impractically large number of trials would be required to make a statistically confident conclusion about fuzzer superiority.

We recognize that conducting thousands of repetitions is infeasible due to computational constraints, and it is NOT our intention to suggest that researchers should simply perform more runs. Instead, our findings highlight the importance of carefully selecting benchmarks that are effective at differentiating fuzzer performance.

To this end, we advocate for the use of pilot experiments prior to large-scale evaluations. Such pilot studies

could help identify which targets have high differential power, i.e. the ability to clearly distinguish the performance of different fuzzers. By first analyzing a small number of exploratory runs across candidate targets, researchers can focus subsequent large-scale experiments on benchmarks that are most informative.

This approach would not only improve the efficiency of fuzzer evaluations but also enhance the reliability of their conclusions.

5.7. Data Availability

All our data, including the sources and the raw data of experiments can be found at https://github.com/BBB-paper/BBB-artifact.

6. Conclusion

We presented a foundational study on the inherent nature of fuzzing, using two common metrics to analyze fuzzer behavior. Through our quantitative approach, we highlight both the challenges of eliminating randomness in fuzzing and the issues present in common fuzzing practices. Building on existing statistical methods, we introduce a technique for estimating the computational effort required to draw statistically significant conclusions when evaluating fuzzers. We hope our research will aid future fuzzing community for more rigorous and reliable fuzzer evaluations.

References

[1] M. Zalewski, "American Fuzzy Lop," https://lcamtuf.coredump.cx/afl/, [Online; accessed March 9, 2026].

[2] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 137–137.

[3] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[4] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya, "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*, New York, NY, USA, 2021.

[5] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, nov 2020. [Online]. Available: https://doi.org/10.1145/3428334

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, aug 2020.

[7] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A framework to build modular and reusable fuzzers," 2022.

[8] D. She, A. Storek, Y. Xie, S. Kweon, P. Srivastava, and S. Jana, "Fox: Coverage-guided fuzzing as online stochastic control," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 765–779.

[9] D. Liu, J. Metzman, M. Bohme, O. Chang, and A. Arya, "Sbft tool competition 2023 - fuzzing track," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 51–54. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SBFT59156.2023.00016

[10] P. Görz, J. Schilling, X. Wu, H. Chen, and R. Gopinath, "Sbft tool competition 2024—fuzzing track," in *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, 2024, pp. 25–28.

[11] Y. Zhang, C. Pang, S. Nagy, X. Chen, and J. Xu, "Profile-guided system optimizations for accelerated greybox fuzzing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1257–1271.

[12] "Bloaty: a size profiler for binaries," https://github.com/google/bloaty, accessed March 9, 2026.

[13] K. Yoshii, P. Llopis, K. Zhang, Y. Luo, S. Ogrenci-Memik, G. Memik, R. Sankaran, and P. Beckman, "Addressing thermal and performance variability issues in dynamic processors," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[14] B. Acun and L. V. Kale, "Mitigating processor variation through dynamic load balancing," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 1073–1076.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[16] Z. Y. Ding and C. L. Goues, "An empirical study of oss-fuzz bugs," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 131–142, 2021.

[17] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[18] Z. Hanusz, J. Tarasinska, and W. Zielinski, "Shapiro–wilk test with known mean," *REVSTAT-statistical Journal*, vol. 14, no. 1, pp. 89–100, 2016.

[19] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, pp. 1–23, 2016.

[20] G. E. Noether, "Sample size determination for some common nonparametric tests," *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 645–647, 1987. [Online]. Available: http://www.jstor.org/stable/2289477

[21] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[22] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 736–747.

[23] M. Böhme, V. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2020, pp. 1–11.

[24] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 475–485. [Online]. Available: https://doi.org/10.1145/3238147.3238176

[25] R. Swiecki, "Honggfuzz," https://github.com/google/honggfuzz, [Online; accessed March 9, 2026].

[26] "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016, [Online; accessed March 9, 2026].

[27] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, sep 2018, [Online; accessed March 9, 2026].

[28] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, aug 2019, pp. 1949–1966.

[29] "llvm-cov - emit coverage information," https://llvm.org/docs/CommandGuide/llvm-cov.html, Accessed March 9, 2026.

[30] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2020, pp. 1–12.

[31] R. A. Matsouaka and R. A. Betensky, "Power and sample size calculations for the wilcoxon–mann–whitney test in the presence of death-censored observations," *Statistics in Medicine*, vol. 34, no. 3, pp. 406–431, 2015.

[32] J. Ounjai, V. Wüstholz, and M. Christakis, "Green fuzzer benchmarking," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1396–1406.